

Informix[®] Guide to SQL

Syntax

Informix Dynamic Server, Version 7.3
Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, Version 8.2
Informix Dynamic Server, Developer Edition, Version 7.3
Informix Dynamic Server, Workgroup Edition, Version 7.3

February 1998
Part No. 000-4367

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025-1032

Copyright © 1981-1998 by Informix Software, Inc. or its subsidiaries, provided that portions may be copyrighted by third parties, as set forth in documentation. All rights reserved.

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by “®,” and in numerous other countries worldwide:

Answers OnLine™; INFORMIX®; Informix®; Illustra™; C-ISAM®; DataBlade®; Dynamic Server™; Gateway™; NewEra™

All other names or marks may be registered trademarks or trademarks of their respective owners.

Documentation Team: Evelyn Eldridge-Diaz, Geeta Karmarkar, Barbara Nomiya, Tom Noronha, Kathy Schaefer, Kami Shahi, Judith Sherwood

RESTRICTED RIGHTS/SPECIAL LICENSE RIGHTS

Software and documentation acquired with US Government funds are provided with rights as follows: (1) if for civilian agency use, with Restricted Rights as defined in FAR 52.227-19; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by negotiated vendor license as prescribed in DFAR 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce the legend.

Table of Contents

Introduction

About This Manual	3
Types of Users	3
Software Dependencies	4
Assumptions About Your Locale.	4
Demonstration Databases	5
New Features	5
New Features in Version 7.3	5
New Features in Version 8.2	7
Documentation Conventions	7
Typographical Conventions	8
Icon Conventions	9
Syntax Conventions	11
Sample-Code Conventions.	15
Additional Documentation	16
On-Line Manuals	16
Printed Manuals	17
Error Message Files	17
Documentation Notes, Release Notes, Machine Notes	18
Related Reading	19
Compliance with Industry Standards	20
Informix Welcomes Your Comments	20

Chapter 1

Overview of SQL Syntax

How to Enter SQL Statements	1-3
How to Enter SQL Comments	1-6
Categories of SQL Statements	1-9
Stored Procedure Statements	1-11
ANSI Compliance and Extensions	1-12

Chapter 2 SQL Statements

ALLOCATE DESCRIPTOR	2-6
ALTER FRAGMENT	2-8
ALTER INDEX	2-34
ALTER TABLE	2-37
BEGIN WORK.	2-67
CLOSE	2-70
CLOSE DATABASE	2-73
COMMIT WORK.	2-75
CONNECT	2-77
CREATE DATABASE	2-89
CREATE EXTERNAL TABLE	2-92
CREATE INDEX	2-106
CREATE PROCEDURE	2-134
CREATE PROCEDURE FROM	2-144
CREATE ROLE	2-146
CREATE SCHEMA	2-148
CREATE SYNONYM	2-151
CREATE TABLE	2-155
CREATE TRIGGER	2-198
CREATE VIEW	2-230
DATABASE.	2-236
DEALLOCATE DESCRIPTOR	2-239
DECLARE	2-241
DELETE	2-258
DESCRIBE	2-262
DISCONNECT	2-267
DROP DATABASE	2-271
DROP INDEX	2-273
DROP PROCEDURE	2-275
DROP ROLE	2-276
DROP SYNONYM	2-277
DROP TABLE	2-279
DROP TRIGGER	2-282
DROP VIEW	2-283
EXECUTE	2-285
EXECUTE IMMEDIATE	2-294
EXECUTE PROCEDURE	2-297
FETCH	2-301
FLUSH	2-312
FREE	2-315
GET DESCRIPTOR	2-318
GET DIAGNOSTICS	2-325
GRANT	2-343
GRANT FRAGMENT	2-360
INFO	2-369
INSERT	2-373
LOAD	2-384

LOCK TABLE	2-391
OPEN	2-394
OUTPUT	2-402
PREPARE	2-404
PUT	2-418
RENAME COLUMN	2-426
RENAME DATABASE	2-428
RENAME TABLE	2-429
REVOKE	2-432
REVOKE FRAGMENT	2-444
ROLLBACK WORK	2-449
SELECT	2-451
SET AUTOFREE	2-503
SET CONNECTION	2-506
SET Database Object Mode	2-513
SET DATASKIP	2-535
SET DEBUG FILE TO	2-538
SET DEFERRED_PREPARE	2-541
SET DESCRIPTOR	2-545
SET EXPLAIN	2-553
SET ISOLATION	2-556
SET LOCK MODE	2-561
SET LOG	2-564
SET OPTIMIZATION	2-566
SET PDQPRIORITY	2-570
SET PLOAD FILE	2-574
SET Residency	2-576
SET ROLE	2-579
SET SCHEDULE LEVEL	2-581
SET SESSION AUTHORIZATION	2-582
SET TRANSACTION	2-585
SET Transaction Mode	2-591
START VIOLATIONS TABLE	2-595
STOP VIOLATIONS TABLE	2-614
UNLOAD	2-616
UNLOCK TABLE	2-621
UPDATE	2-623
UPDATE STATISTICS	2-635
WHENEVER	2-646

Chapter 3 SPL Statements

CALL	3-4
CONTINUE	3-7
DEFINE	3-8
EXIT	3-16
FOR	3-18
FOREACH	3-23
IF	3-27

LET	3-31
ON EXCEPTION	3-34
RAISE EXCEPTION	3-39
RETURN	3-41
SYSTEM	3-44
TRACE	3-47
WHILE	3-50

Chapter 4

Segments

Condition	4-5
Database Name	4-22
Database Object Name	4-25
Data Type	4-27
DATETIME Field Qualifier	4-32
Expression	4-34
Identifier	4-114
INTERVAL Field Qualifier	4-131
Literal DATETIME	4-134
Literal INTERVAL	4-137
Literal Number	4-140
Optimizer Directives	4-142
Owner Name	4-155
Quoted String	4-158
Relational Operator	4-162

Index

Introduction

About This Manual.	3
Types of Users	3
Software Dependencies	4
Assumptions About Your Locale.	4
Demonstration Databases	5
New Features.	5
New Features in Version 7.3	5
New Features in Version 8.2	6
Documentation Conventions	7
Typographical Conventions	8
Icon Conventions	8
Comment Icons	9
Feature, Product, and Platform Icons	9
Compliance Icons	10
Syntax Conventions	11
Elements That Can Appear on the Path	12
How to Read a Syntax Diagram	14
Sample-Code Conventions	15
Additional Documentation	16
On-Line Manuals	16
Printed Manuals	16
Error Message Files	17
Documentation Notes, Release Notes, Machine Notes	17
Related Reading	18
Compliance with Industry Standards	19
Informix Welcomes Your Comments.	19

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

This manual is a companion volume to the *Informix Guide to SQL: Reference*, the *Informix Guide to SQL: Tutorial*, and the *Informix Guide to Database Design and Implementation*.

This manual contains all the syntax descriptions for Structured Query Language (SQL) and Stored Procedure Language (SPL) statements. The *Informix Guide to Database Design and Implementation* explains the philosophy and concepts behind relational databases, the *Informix Guide to SQL: Reference* provides reference information for aspects of SQL other than the language statements, and the *Informix Guide to SQL: Tutorial* gives introductory examples as well as extended discussions of various aspects of SQL.

Types of Users

This manual is for the following users:

- Database-application programmers
- Database administrators
- Database users

This manual assumes that you have the following background:

- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming

If you have limited experience with relational databases, SQL, or your operating system, refer to the [Getting Started](#) manual for your database server for a list of supplementary titles.

Software Dependencies

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

- Informix Dynamic Server, Version 7.3
- Dynamic Server with AD and XP Options, Version 8.2
- Informix Dynamic Server, Developer Edition, Version 7.3
- Informix Dynamic Server, Workgroup Edition, Version 7.3

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.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the [Informix Guide to GLS Functionality](#).

Demonstration Databases

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. You can use SQL scripts provided with DB-Access to derive a second database, called **sales_demo**. This database illustrates a dimensional schema for data-warehousing applications. Sample command files are also included for creating and populating these databases.

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 the *[Informix Guide to SQL: Reference](#)*.

The scripts that you use to install the demonstration databases reside in the **\$INFORMIXDIR/bin** directory on UNIX platforms and the **%INFORMIXDIR%\bin** directory on Windows NT platforms. For a complete explanation of how to create and populate the **stores7** demonstration database, refer to the *[DB-Access User Manual](#)*. For an explanation of how to create and populate the **sales_demo** database, refer to the *[Informix Guide to Database Design and Implementation](#)*.

New Features

The following sections describe new database server features relevant to this manual. For a comprehensive list of new features, see the release notes for your database server.

New Features in Version 7.3

Most of the new features for Version 7.3 of Informix Dynamic Server fall into five major areas:

- Reliability, availability, and serviceability
- Performance

- Windows NT-specific features
- Application migration
- Manageability

Several additional features affect connectivity, replication, and the optical subsystem.

This manual includes information about the following new features:

- Performance:
 - Enhancements to the SELECT statement to allow selection of the first *n* rows
 - New statement: SET Residency
 - Enhancements to the SET OPTIMIZATION statement
 - New syntax for optimizer directives
- Application migration:
 - New functions for case-insensitive search (UPPER, LOWER, INITCAP)
 - New functions for string manipulations (REPLACE, SUBSTR, LPAD, RPAD)
 - New CASE expression
 - New NVL and DECODE functions
 - New date-conversion functions (TO_CHAR and TO_DATE)
 - New options for the DBINFO function
 - Enhancements to the CREATE VIEW, DESCRIBE, and EXECUTE PROCEDURE statements

New Features in Version 8.2

This manual describes the following new features that have been implemented in Version 8.2 of Dynamic Server with AD and XP Options:

- Enhanced fragmentation of table data
- Support for query scheduling
- Enhanced indexes to support data-warehousing applications
- Global language support (GLS)
- New aggregates: standard deviation, range, and variance
- Enhanced optimizer reports
- Enhancements to the CREATE VIEW statement

This manual also discusses the following features, which were introduced in Version 8.1 of Dynamic Server with AD and XP Options:

- New join methods for use across multiple computers
- The CASE expression in certain Structured Query Language (SQL) statements
- Nonlogging tables
- External tables for high-performance loading and unloading

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.
<i>italics</i>	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.
<code>monospace</code>	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 within a table or section.
→	This symbol indicates a menu item. For example, “Choose Tools→Options ” means choose the Options item from the Tools menu.



Tip: When you are instructed to “enter” characters or to “execute” a command, immediately press RETURN after you type the indicated information on your keyboard. When you are instructed to “type” the text or to “press” other keys, you do not need to press RETURN.

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 <i>warning</i> icon identifies vital instructions, cautions, or critical information.
	The <i>important</i> icon identifies significant information about the feature or operation that is being described.
	The <i>tip</i> icon identifies additional details or shortcuts for the functionality that is being described.

Feature, Product, and Platform Icons

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

Icon	Description
	Identifies information that is specific to Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options.
	Identifies information that is specific to DB-Access.
	Identifies information that is specific to INFORMIX-ESQL / C.
	Identifies information that relates to the Informix Global Language Support (GLS) feature.
	Identifies information that is specific to Dynamic Server and its editions. In some cases, the identified section applies only to Informix Dynamic Server and not to Informix Dynamic Server, Workgroup and Developer Editions. Such information is clearly identified.
	Identifies information that is specific to UNIX platforms.
	Identifies information that is specific to SQL Editor, which is a component of Informix Enterprise Command Center for Informix Dynamic Server.
	Identifies information that is specific to Informix Dynamic Server, Workgroup and Developer Editions.
	Identifies information that is specific to the Windows NT environment.

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

Compliance Icons

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

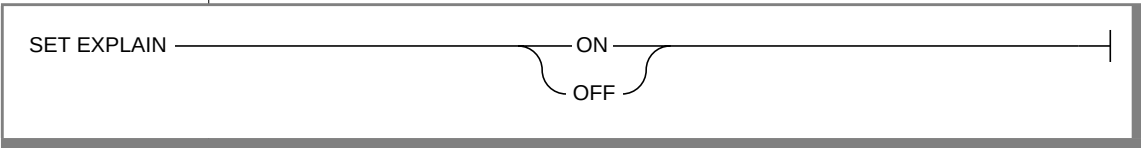
Icon	Description
	Identifies information that is specific to an ANSI-compliant database.
	Identifies information that is an Informix extension to ANSI SQL-92 entry-level standard SQL.
	Identifies functionality that conforms to X/Open.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. If an icon appears next to a section heading, the compliance information ends at the next heading at the same or higher level. A ♦ symbol indicates the end of compliance information that appears in a table row or a set of paragraphs within a section.

Syntax Conventions

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

Figure 1
Example of a Simple Syntax Diagram



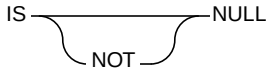
Each syntax diagram begins at the upper-left corner and ends at the upper-right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement.

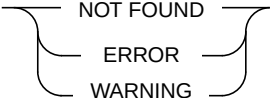
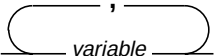
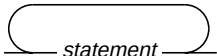
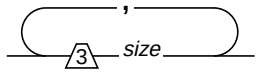
Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. The path always approaches elements from the left and continues to the right, except in the case of separators in loops. For separators in loops, the path approaches counterclockwise from the 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.
<i>variable</i>	A word in <i>italics</i> 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.
Back to ADD Clause p. 1-14	A reference in a box in the upper right hand corner of a subdiagram refers to the next higher-level diagram of which this subdiagram is a member.
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.

Element	Description
	These icons might appear in a syntax diagram:
	This path is valid only for DB-Access.
	This path is valid only for INFORMIX-ESQL / C.
	This path is valid only if you are using Informix Stored Procedure Language (SPL).
	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. The Informix extension warnings tend to be conservative. Sometimes the warnings appear even when a syntax path conforms to the ANSI standard.
	This path is valid only if your database or application uses a nondefault GLS locale.
	A shaded option is the default action.
	Syntax that is enclosed between a pair of arrows is a subdiagram.
	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.)

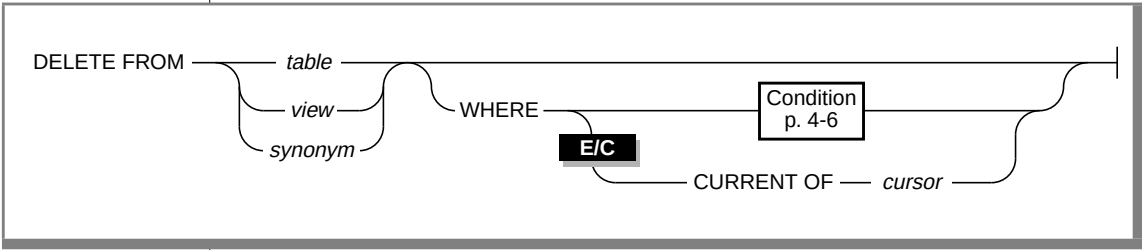
Element	Description
	A set of multiple branches indicates that a choice among more than two different paths is available.
 	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items. If no symbol appears, a blank space is the separator.
	A gate () 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



To use this 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. You can delete a table, view, or synonym:
 - Follow the diagram by typing the table name, view name, or synonym, as desired.
 - You can type the keyword `WHERE` to limit the rows that are deleted.
 - If you specify the keyword `WHERE` and you are using DB-Access or the SQL Editor, you must include the Condition clause to specify a condition to delete. To find the syntax for specifying a condition, go to the “Condition” segment on the specified page.
 - If you are using ESQL/C, you can include either the Condition clause to delete a specific condition or the `CURRENT OF cursor` clause to delete a row from the table.
3. Follow the diagram to the terminator. Your `DELETE` statement is complete.

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 ESQL/C, you must use EXEC SQL at the start of each statement and a semicolon 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 product, 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
- Related reading

On-Line Manuals

An Answers OnLine 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 the installation insert that accompanies Answers OnLine.

Printed Manuals

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com. Please provide the following information when you place your order:

- 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 of the Informix error messages and their corrective actions. For a detailed description of these error messages, refer to *Informix Error Messages* in Answers OnLine.

To read the error messages under UNIX, you can use the following commands.

Command	Description
finderr	Displays error messages on line
rofferr	Formats error messages for printing

◆

To read error messages and corrective actions under Windows NT, use the **Informix Find Error** utility. To display this utility, choose **Start→Programs→Informix** from the Task Bar. ◆

UNIX

WIN NT

UNIX

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following sections describe the on-line files that supplement the information in this manual. Please examine these files before you begin using your database server. They contain vital information about application and performance issues.

On UNIX platforms, the following on-line files appear in the **\$INFORMIXDIR/release/en_us/0333** directory.

On-Line File	Purpose
SQLSDOC_x.y	The documentation-notes file for your version of this manual describes features that are not covered in the manual or that have been modified since publication. Replace x.y in the filename with the version number of your database server to derive the name of the documentation-notes file for this manual.
SERVERS_x.y	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. Replace x.y in the filename with the version number of your database server to derive the name of the release-notes file.
IDS_x.y	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. Replace x.y in the filename with the version number of your database server to derive the name of the machine-notes file.

◆

WIN NT

The following items appear in the Informix folder. To display this folder, choose **Start→Programs→Informix** from the Task Bar.

Item	Description
Documentation Notes	This item includes additions or corrections to manuals, along with information about features that may not be covered in the manuals or that have been modified since publication.
Release Notes	This item 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.

Machine notes do not apply to Windows NT platforms. ♦

Related Reading

The following publications provide additional information about the topics that are discussed in this manual. For a list of publications that provide an introduction to database servers and operating-system platforms, refer to the [Getting Started](#) manual.

- *A Guide to the SQL Standard* by C. J. Date with H. Darwen (Addison-Wesley Publishing, 1993)
- *Understanding the New SQL: A Complete Guide* by J. Melton and A. Simon (Morgan Kaufmann Publishers, 1993)
- *Using SQL* by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

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. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

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:

650-926-6571

We appreciate your feedback.

Overview of SQL Syntax

How to Enter SQL Statements	1-3
How to Enter SQL Comments	1-6
Categories of SQL Statements	1-9
Stored Procedure Statements	1-11
ANSI Compliance and Extensions	1-12

T

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

Section	Starting Page	Scope
“How to Enter SQL Statements”	1-3	This section shows how to use the statement descriptions to enter SQL statements correctly.
“How to Enter SQL Comments”	1-6	This section shows how to enter comments for SQL statements.
“Categories of SQL Statements”	1-9	This section lists SQL statements by functional category.
“ANSI Compliance and Extensions”	1-12	This section lists SQL statements by degree of ANSI compliance.

How to Enter SQL Statements

The purpose of the statement descriptions in this manual is to help you to enter SQL statements successfully. 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 include:

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

Using Syntax Diagrams and Syntax Tables

Before you try to use the syntax diagrams in this chapter, it is helpful to read the [“Syntax Conventions” on page 11](#) 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 points 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 15](#) 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	ESQL/C	Stored Procedures	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. If you want to use the double dash 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

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

SPL

You can use either comment symbol in any line of a stored procedure. For further information, see “[Adding Comments to the Procedure](#)” on [page 2-139](#). Also see the discussion of how to comment and document a procedure in the [Informix Guide to SQL: Tutorial](#). ♦

E/C

In ESQL/C, you can use the double dash (--) to comment SQL statements. For further information on the use of SQL comment symbols and language-specific comment symbols in ESQL/C programs, see the [INFORMIX-ESQL/C Programmer's Manual](#). ♦

Examples of SQL Comment Symbols

Some simple examples can help to illustrate the different ways to use 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
```

DB

SPL

Examples of the Curly-Brackets Symbols

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}
```

GLS

Non-ASCII Characters in SQL Comments

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 [Informix Guide to GLS Functionality](#).

Categories of SQL Statements

SQL statements are divided into the following categories:

- Data definition statements
- Data manipulation statements
- Cursor manipulation statements
- Cursor optimization statements
- Dynamic management statements
- Data access statements
- Data integrity statements
- Optimization statements
- Stored procedure statements
- Auxiliary statements
- Client/server connection statements
- Optical subsystem statements

The specific statements for each category are as follows.

Data Definition Statements

ALTER FRAGMENT	CREATE VIEW
ALTER INDEX	DATABASE
ALTER TABLE	DROP DATABASE
CLOSE DATABASE	DROP INDEX
CREATE DATABASE	DROP PROCEDURE
CREATE EXTERNAL TABLE	DROP ROLE
CREATE INDEX	DROP SYNONYM
CREATE PROCEDURE	DROP TABLE
CREATE PROCEDURE FROM	DROP TRIGGER
CREATE ROLEDCREATE SCHEMA	DROP VIEW
CREATE SYNONYM	RENAME COLUMN
CREATE TABLE	RENAME DATABASE
CREATE TRIGGER	RENAME TABLE

Data Manipulation Statements

DELETE
INSERT
LOAD

SELECT
UNLOAD
UPDATE

Cursor Manipulation Statements

CLOSE
DECLARE
FETCH
FLUSH

FREE
OPEN
PUT
SET AUTOFREE

Cursor Optimization Statements

SET AUTOFREE
SET DEFERRED_PREPARE

Dynamic Management Statements

ALLOCATE DESCRIPTOR
DEALLOCATE DESCRIPTOR
DESCRIBE
EXECUTE
EXECUTE IMMEDIATE

FREE
GET DESCRIPTOR
PREPARE
SET DEFERRED_PREPARE
SET DESCRIPTOR

Data Access Statements

GRANT
GRANT FRAGMENT
LOCK TABLE
REVOKE
REVOKE FRAGMENT
SET ISOLATION

SET LOCK MODE
SET ROLE
SET SESSION AUTHORIZATION
SET TRANSACTION
UNLOCK TABLE

Data Integrity Statements

BEGIN WORK
COMMIT WORK
ROLLBACK WORK
SET DATABASE OBJECT MODE
SET LOG

SET PLOAD FILE
SET TRANSACTION MODE
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE

Optimization Statements

SET EXPLAIN
SET OPTIMIZATION
SET PDQPRIORITY

SET Residency
SET SCHEDULE LEVEL
UPDATE STATISTICS

Stored Procedure Statements

EXECUTE PROCEDURE
SET DEBUG FILE TO

Auxiliary Statements

INFO
OUTPUT

GET DIAGNOSTICS
WHENEVER

Client/Server Connection Statements

CONNECT
DISCONNECT

SET CONNECTION

Optical Subsystem Statements

ALTER OPTICAL CLUSTER
CREATE OPTICAL CLUSTER
DROP OPTICAL CLUSTER

RELEASE
RESERVE
SET MOUNTING TIMEOUT

IDS



Important: Optical Subsystem statements are described in the “[Guide to the Optical Subsystem](#).” Optical Subsystem statements are not available with Dynamic Server, Workgroup and Developer Editions.

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 Extension

CREATE SCHEMA AUTHORIZATION
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 DESCRIPTOR	INFO
ALTER FRAGMENT	LOAD
ALTER INDEX	LOCK TABLE
ALTER OPTICAL CLUSTER	OUTPUT
ALTER TABLE	PREPARE
BEGIN WORK	PUT
CLOSE DATABASE	RELEASE
CONNECT	RENAME COLUMN
CREATE DATABASE	RENAME DATABASE
CREATE EXTERNAL TABLE	RENAME TABLE
CREATE INDEX	RESERVE
CREATE OPTICAL CLUSTER	REVOKE
CREATE PROCEDURE	REVOKE FRAGMENT
CREATE PROCEDURE FROM	SET
CREATE ROLE	SET AUTOFREE
CREATE SYNONYM	SET DATASKIP
CREATE TRIGGER	SET DEBUG FILE TO
DATABASE	SET DEFERRED_PREPARE
DEALLOCATE DESCRIPTOR	SET DESCRIPTOR
DESCRIBE	SET EXPLAIN
DISCONNECT	SET ISOLATION
DROP DATABASE	SET LOCK MODE
DROP INDEX	SET LOG
DROP OPTICAL CLUSTER	SET MOUNTING TIMEOUT
DROP PROCEDURE	SET OPTIMIZATION
DROP ROLE	SET PDQPRIORITY
DROP SYNONYM	SET PLOAD FILE
DROP TABLE	SET RESIDENCY
DROP TRIGGER	SET ROLE
DROP VIEW	SET SCHEDULE LEVEL
EXECUTE IMMEDIATE	SET TRANSACTION
EXECUTE PROCEDURE	START VIOLATIONS TABLE
FLUSH	STOP VIOLATIONS TABLE
FREE	UNLOAD
GET DESCRIPTOR	UNLOCK TABLE
GET DIAGNOSTICS	UPDATE STATISTICS
GRANT FRAGMENT	

SQL Statements

ALLOCATE DESCRIPTOR	2-6
ALTER FRAGMENT	2-8
ALTER INDEX	2-34
ALTER TABLE	2-37
BEGIN WORK	2-67
CLOSE	2-70
CLOSE DATABASE	2-73
COMMIT WORK	2-75
CONNECT	2-77
CREATE DATABASE	2-89
CREATE EXTERNAL TABLE	2-92
CREATE INDEX	2-106
CREATE PROCEDURE	2-134
CREATE PROCEDURE FROM	2-144
CREATE ROLE	2-146
CREATE SCHEMA	2-148
CREATE SYNONYM	2-151
CREATE TABLE	2-155
CREATE TRIGGER	2-198
CREATE VIEW	2-230
DATABASE	2-236
DEALLOCATE DESCRIPTOR	2-239
DECLARE	2-241
DELETE	2-258
DESCRIBE	2-262
DISCONNECT	2-267
DROP DATABASE	2-271
DROP INDEX	2-273
DROP PROCEDURE	2-275
DROP ROLE	2-276
DROP SYNONYM	2-277

DROP TABLE	2-279
DROP TRIGGER	2-282
DROP VIEW	2-283
EXECUTE	2-285
EXECUTE IMMEDIATE	2-294
EXECUTE PROCEDURE	2-297
FETCH	2-301
FLUSH	2-312
FREE	2-315
GET DESCRIPTOR	2-318
GET DIAGNOSTICS	2-325
GRANT	2-342
GRANT FRAGMENT	2-359
INFO	2-368
INSERT	2-372
LOAD	2-383
LOCK TABLE	2-390
OPEN	2-393
OUTPUT	2-401
PREPARE	2-403
PUT	2-417
RENAME COLUMN	2-425
RENAME DATABASE	2-427
RENAME TABLE	2-428
REVOKE	2-431
REVOKE FRAGMENT	2-443
ROLLBACK WORK	2-448
SELECT	2-450
SET AUTOFREE	2-502
SET CONNECTION	2-505
SET Database Object Mode	2-512
SET DATASKIP	2-534
SET DEBUG FILE TO	2-537
SET DEFERRED_PREPARE	2-540
SET DESCRIPTOR	2-544
SET EXPLAIN	2-552
SET ISOLATION	2-555
SET LOCK MODE	2-560
SET LOG	2-563
SET OPTIMIZATION	2-565
SET PDQPRIORITY	2-569

SET PLOAD FILE	.2-573
SET Residency.	.2-575
SET ROLE	.2-578
SET SCHEDULE LEVEL	.2-580
SET SESSION AUTHORIZATION	.2-581
SET TRANSACTION	.2-584
SET Transaction Mode	.2-590
START VIOLATIONS TABLE	.2-594
STOP VIOLATIONS TABLE	.2-613
UNLOAD	.2-615
UNLOCK TABLE.	.2-620
UPDATE.	.2-622
UPDATE STATISTICS	.2-634
WHENEVER	.2-645

This chapter gives comprehensive reference descriptions of SQL statements. The statement descriptions appear in alphabetical order. For an explanation of the structure of statement descriptions, see [Chapter 1, “Overview of SQL Syntax.”](#)

+

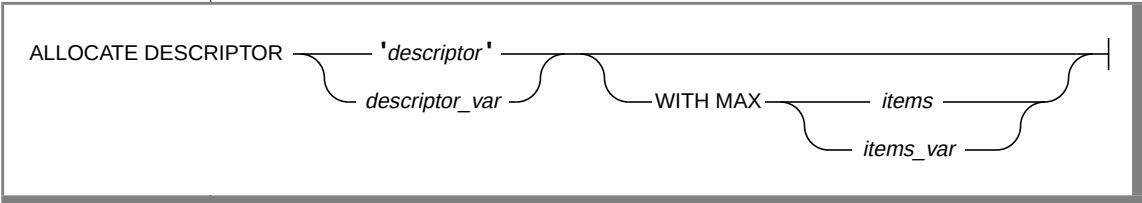
E/C

ALLOCATE DESCRIPTOR

Use the ALLOCATE DESCRIPTOR statement to allocate memory for a system-descriptor area. This statement creates a place in memory to hold information that a DESCRIBE statement obtains or to hold information about the WHERE clause of a statement.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>descriptor</i>	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. 4-157
<i>descriptor_var</i>	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.
<i>items</i>	Number of item descriptors in the system-descriptor area The default value is 100.	Value must be unsigned INTEGER.	Literal Number, p. 4-139
<i>items_var</i>	Host variable that contains the number of <i>items</i>	Data type must be INTEGER or SMALLINT.	Name must conform to language-specific rules for variable names.

Usage

The `ALLOCATE DESCRIPTOR` statement creates a system-descriptor area. 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.

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

WITH MAX Clause

You can use the `WITH MAX` clause to indicate the number of item descriptors you need. When you use this clause, the `COUNT` field is set to the number of *items* you specify. This number must be greater than zero. If you do not specify the `WITH MAX` clause, the default value of the `COUNT` field is 100.

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

```
EXEC SQL allocate descriptor :descname with max :occ;
EXEC SQL allocate descriptor 'desc1' with max 3;
```

References

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

For more information on system-descriptor areas, refer to the [*INFORMIX-ESQL/C Programmer's Manual*](#).

+

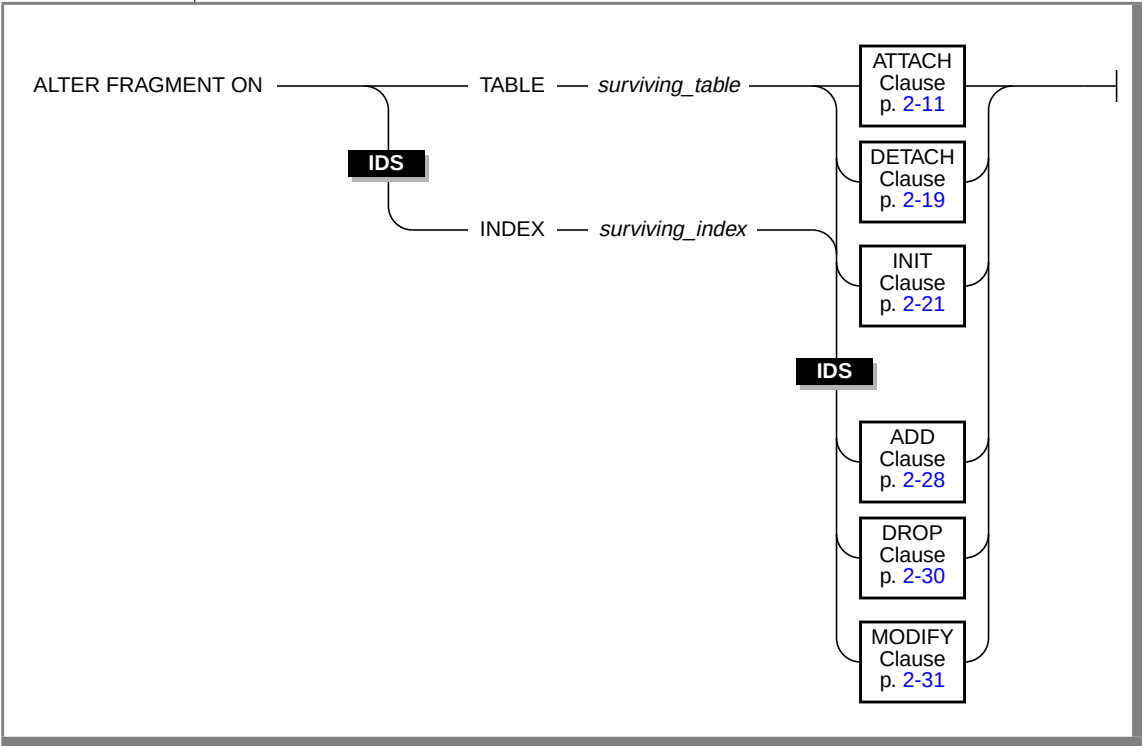
W/D

ALTER FRAGMENT

Use the ALTER FRAGMENT statement to alter the distribution strategy or storage location of an existing table or index as well as to create fragments initially.

This statement is not available with Dynamic Server, Workgroup and Developer Editions. ♦

Syntax



Element	Purpose	Restrictions	Syntax
<i>surviving_index</i>	Index on which you execute the ALTER FRAGMENT statement	The index must exist at the time you execute the statement.	Database Object Name, p. 4-25
<i>surviving_table</i>	Table on which you execute the ALTER FRAGMENT statement	The table must exist at the time you execute the statement.	Database Object Name, p. 4-25

Usage

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 or slice from a fragmentation strategy and places it in a new table.
INIT	Provides the following options: ■ Define and initialize a fragmentation strategy on a table. ■ Create a fragmentation strategy for tables. ■ Change the order of evaluation of fragment expressions. ■ Alter the fragmentation strategy of an existing table or index. ■ Change the storage location of an existing table.
ADD	Adds an additional fragment to an existing fragmentation list.
DROP	Drops an existing fragment from a fragmentation list.
MODIFY	Changes an existing fragmentation expression.

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

In Dynamic Server with AD and XP Options, you cannot use the ALTER FRAGMENT statement on a generalized-key (GK) index. Also, you cannot use the ALTER FRAGMENT statement on any table that has a dependent GK index defined on it. ♦

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.

The ALTER FRAGMENT statement applies only to table or index fragments that are located at the current site (or cluster, for Dynamic Server with AD and XP Options). No remote information is accessed or updated.

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

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.
- 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, see your [Administrator's Guide](#). That guide also contains detailed instructions about how to turn off logging. For information about backups, refer to your [Backup and Restore Guide](#).

Determining the Number of Rows in the Fragment

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

To find out how many rows are in a fragment

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

AD/XP

Modifying Hybrid-Fragmented Tables

In Dynamic Server with AD and XP Options, when you modify a hybrid-fragmented table with the ATTACH and DETACH clauses, the unit of storage is the set of dbspaces associated with a given expression in the base fragmentation strategy of the hybrid-fragmented table. The dbspace name is a *derived dbspace identifier* formed by appending the relative dbspace number to the dbslice name. You can use .1 for the relative dbspace number if the table was created with that dbslice. For a description of hybrid-fragmented tables and derived dbspace identifiers, refer to the [Informix Guide to Database Design and Implementation](#).

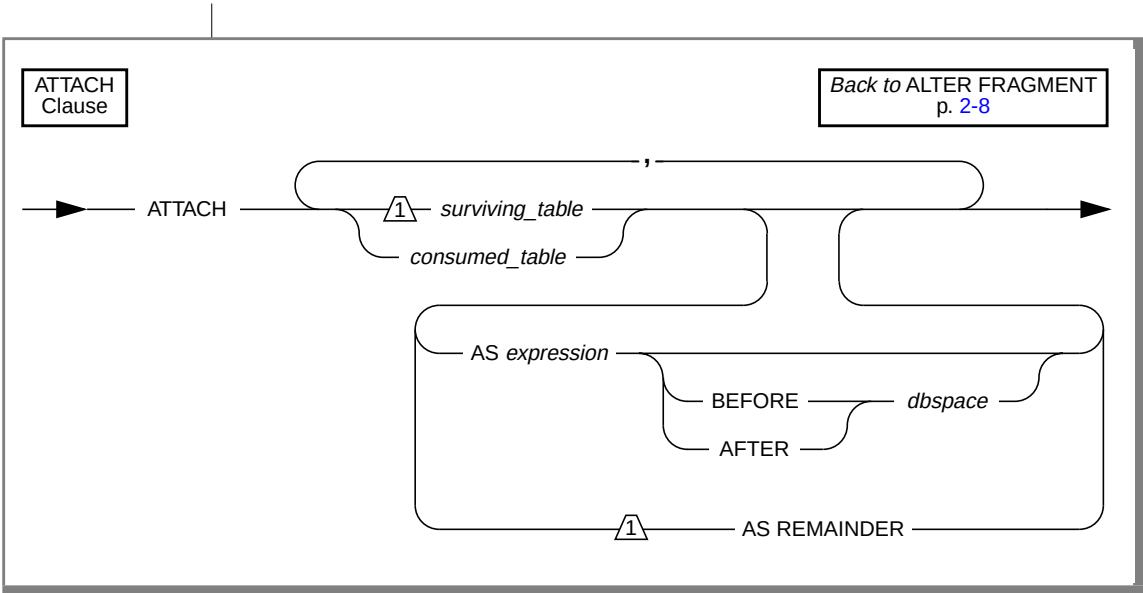
ATTACH Clause

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



ALTER FRAGMENT



Element	Purpose	Restrictions	Syntax
<i>consumed_table</i>	Nonfragmented table on which you execute the ATTACH clause In Dynamic Server with AD and XP Options, <i>consumed_table</i> can also be a table with hash fragmentation.	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.	Database Object Name, p. 4-25
<i>dbspace</i>	Dbpace 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. 4-113

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>expression</i>	Expression that defines a fragment	The <i>expression</i> 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 <i>expression</i>.	Condition, p. 4-5, and Expression, p. 4-33
<i>surviving_table</i>	Table on which you execute the ATTACH clause 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.	Database Object Name, p. 4-25

(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 you attach the tables, all consumed tables that were specified in the ATTACH clause no longer exist. You must reference the records that were in the consumed tables through the surviving table.

All consumed tables listed in the ATTACH clause must be identical in structure to the surviving table; that is, all column definitions must match. The number, names, data types, and relative position of the columns must be identical. You cannot attach tables that contain serial columns. Indexes and views on the surviving table survive the ATTACH, but indexes and views on the consumed table are dropped.

In Dynamic Server, triggers on the surviving table survive the ATTACH, but triggers on the consumed table are dropped. Triggers are not activated with the ATTACH clause. ♦

IDS

Using the ATTACH Clause

In Dynamic Server with AD and XP Options, when you create or append to a hybrid-fragmented table, the positioning specification (BEFORE, AFTER, or REMAINDER) applies to an entire slice. You can use any dbspace in a slice to identify the slice for the BEFORE or AFTER position.

In addition to an identical structure, every consumed table must be of the same type as the surviving table. For information about how to specify the type of a table, refer to CREATE TABLE.

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

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

Restrictions on the ATTACH Clause

You cannot use the ATTACH clause in the certain situations. The attach operation fails:

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

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 in the *attach list*. The attach list is the list of tables in the ATTACH clause.

In Dynamic Server, if you want the newly-created single, fragmented table to include a rowid column, 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. In the following example, a round-robin distribution scheme fragments the table **cur_acct**, and the table **old_acct** is a nonfragmented table that resides in a separate dbspace. The example shows how to attach **old_acct** to **cur_acct**:

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

BEFORE and AFTER Clauses

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

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

Using ATTACH to Fragment Tables: Hybrid Fragmentation

In Dynamic Server with AD and XP Options, you can use the BEFORE and AFTER clauses when you attach a hash-fragmented table to a hybrid-fragmented table (or to another hash-fragmented table, thereby creating a hybrid-fragmented table). In this case, the BEFORE and AFTER clauses allow you to place the set of fragments (called a slice) associated with a given expression before or after the fragments associated with some other fragmentation expression.

You can specify the dbspace name of any fragment in a slice in the BEFORE or AFTER clause; the database server identifies the slice that contains that dbspace and places attaching fragments accordingly.

Hybrid Fragmentation Example

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

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

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

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


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

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

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 alter the fragment, you see that table **cur_acct** is fragmented by expression into three dbspaces. For additional information about the **sysfragments** system catalog table, see the [Informix Guide to SQL: Reference](#).

In addition to simple range rules, you can use the ATTACH clause to fragment by expression with hash or arbitrary rules. For a discussion of all types of expressions in an expression-based distribution scheme, see “[FRAGMENT BY Clause for Tables](#)” on page 2-22.

What Happens to Indexes?

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

When you attach tables, any indexes that are defined on the consumed table no longer exist, and all rows in the consumed table (**new_acct**) are subject to the indexes that are defined in the surviving table (**cur_acct**).

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.

What Happens to BYTE and TEXT Columns?

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

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

IDS

AD/XP

What Happens to Triggers?

In Dynamic Server, when you attach tables, any triggers that are defined on the consumed table no longer exist, and all rows in the consumed table (**new_acct**) are subject to the 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 activate triggers.

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 2-22](#).

DETACH
Clause
[Back to ALTER FRAGMENT
p. 2-8](#)

→ DETACH dbspace new_table →

Element	Purpose	Restrictions	Syntax
<i>dbspace</i>	Dbspace that contains the fragment to be detached	The dbspace must exist when you execute the statement.	Identifier, p. 4-113
<i>new_table</i>	Table that results after you execute the ALTER FRAGMENT statement	The table must not exist before you execute the statement.	Database Object Name, p. 4-25

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 new table that results from the execution of the DETACH clause does not inherit any privileges from the original table. Instead this table has the default privileges for any new table. For further information on default table-level privileges, see the GRANT statement on [page 2-342](#).

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.

AD/XP

Using the DETACH Clause

In Dynamic Server with AD and XP Options, you cannot use the DETACH clause if the table has a dependent GK index defined on it.

If the surviving table of a DETACH operation has hybrid fragmentation and the detached slice had more than a single fragment, the new table has hash fragmentation. In a hybrid-fragmented table, you specify the slice to be detached by naming *any* dbspace in that slice.

DETACH Clause Example

Consider the **sales_info** table discussed in the [“Hybrid Fragmentation Example” on page 2-16](#). Once the January 1997 data is available in the **sales_info** table, you might archive year-old **sales_info** data.

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

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

IDS

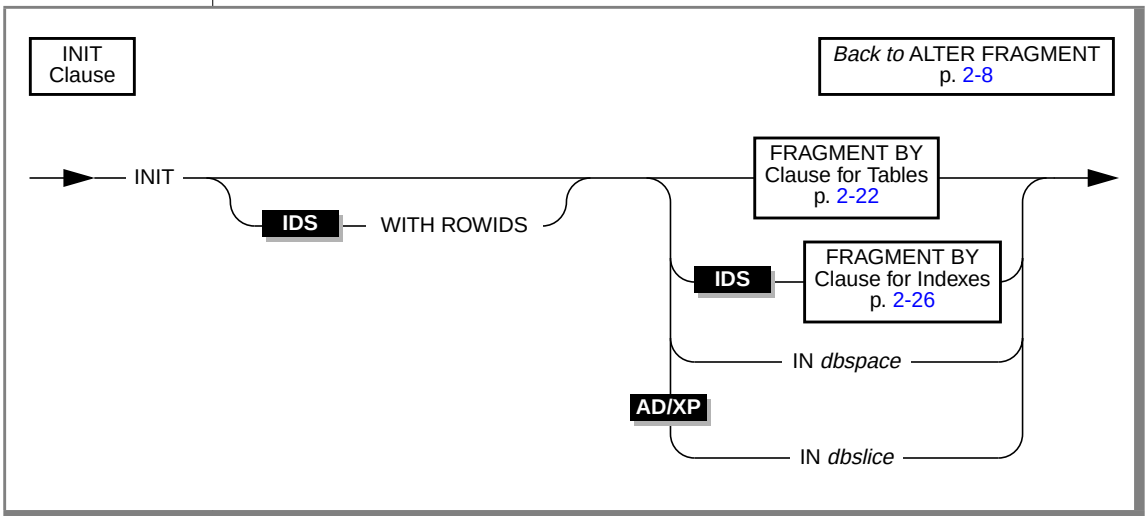
INIT Clause

The INIT clause allows you to fragment an existing table that is not fragmented without redefining the table, to convert a fragmented table to a nonfragmented table, or to move an unfragmented table from one dbspace to another dbspace.

In Dynamic Server, the INIT clause also allows you to fragment an existing index that is not fragmented without redefining the index, or to convert a fragmented index to a nonfragmented index. ♦

With the INIT clause, you can also convert an existing fragmentation strategy to another fragmentation strategy. Any existing fragmentation strategy is discarded, and records are moved to fragments as defined in the new fragmentation strategy.

Using the INIT clause to modify a table changes the **tabid** value in system catalog tables for the affected table. It also changes the **constrid** of all unique and referential constraints on the table.



Element	Purpose	Restrictions	Syntax
<i>dbslice</i>	Dbslice that contains the fragmented information	The dbslice must exist at the time you execute the statement.	Identifier, p. 4-113
<i>dbspace</i>	Dbpace that contains the fragmented information	The dbspace must exist at the time you execute the statement.	Identifier, p. 4-113

IDS

In Dynamic Server, 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. ♦

AD/XP

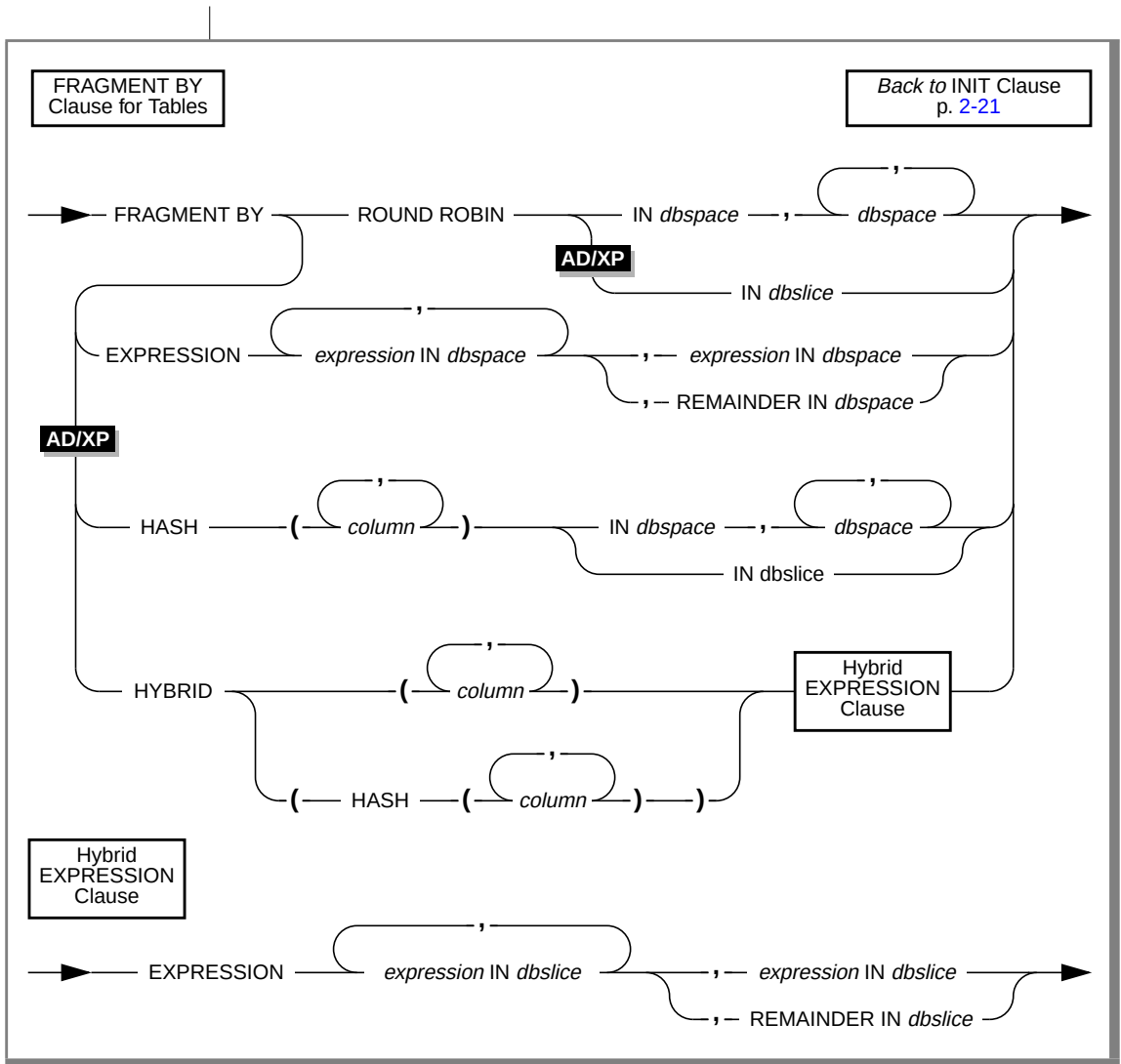
In Dynamic Server with AD and XP Options, when you use the INIT clause to fragment an existing nonfragmented table, indexes retain their existing fragmentation strategy.

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

FRAGMENT BY Clause for Tables

The INIT FRAGMENT BY clause for tables allows you to:

- fragment an existing non-fragmented table without redefining the table.
- convert an existing fragmentation strategy to another fragmentation strategy. Any existing fragmentation strategy is discarded, and records are moved to fragments as defined in the new fragmentation strategy.
- convert a fragmented table to a nonfragmented table.



ALTER FRAGMENT

Element	Purpose	Restrictions	Syntax
<i>column</i>	Column that contains information to be fragmented	The column must exist.	Identifier, p. 4-113
<i>dbspace</i>	Dbospace that contains the fragmented information	You must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces.	Identifier, p. 4-113
<i>dbslice</i>	Dbslice that contains fragmented information	The dbslice must be defined.	Identifier, p. 4-113
<i>expression</i>	Expression that defines a fragment using a range, hash, or arbitrary rule	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 <i>expression</i> .	Condition, p. 4-5 , and Expression, p. 4-33

Changing an Existing Fragmentation Strategy on a Single Table

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 an ALTER FRAGMENT statement that redefines the fragmentation strategy:

```
CREATE TABLE account (col1 int, col2 int)
    FRAGMENT BY ROUND ROBIN IN dbbsp1, dbbsp2;

ALTER FRAGMENT ON TABLE account
    INIT FRAGMENT BY EXPRESSION
    col1 < 0 in dbbsp1,
    col2 >= 0 in dbbsp2;
```

If any existing dbspaces are full when you redefine a fragmentation strategy, you must fragment the table into different dbspaces than the full dbspaces.

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. Similarly, system indexes that do not use user indexes become nonfragmented indexes and are moved from the database dbspace to the table dbspace. If any system indexes use detached user indexes, the system indexes are not affected when you use 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;
```

IDS



IDS

WITH ROWIDS Clause

In Dynamic Server, 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. Each row contains an additional 4 bytes to store the **rowid** column after you add the WITH ROWIDS clause.

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

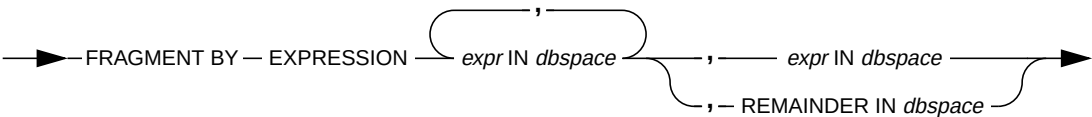
FRAGMENT BY Clause for Indexes

In Dynamic Server, the INIT FRAGMENT BY clause for indexes allows you to fragment an existing index that is not fragmented without redefining the index. You can convert an existing fragmentation strategy to another fragmentation strategy. Any existing fragmentation strategy is discarded, and records are moved to fragments as defined in the new fragmentation strategy. You can also convert a fragmented index to a nonfragmented index.

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 and arbitrary rules. For an explanation of these rules, see [“FRAGMENT BY Clause for Tables” on page 2-22](#).

FRAGMENT BY
Clause for Indexes

Back to INIT Clause
p. 2-21



Element	Purpose	Restrictions	Syntax
<i>dbspace</i>	Dbspace that contains the fragmented information	You must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces.	Identifier, p. 4-113
<i>expression</i>	Expression that defines a fragment using a range, hash, or arbitrary rule	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 and /or time functions, or aggregates are allowed in <i>expression</i> .	Condition, p. 4-5, and Expression, p. 4-33

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.

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

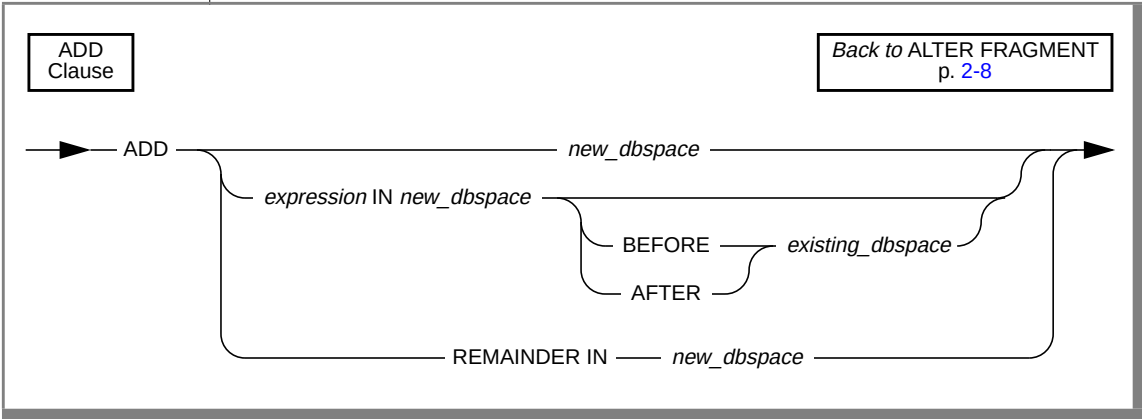
System indexes (such as those used in referential constraints and unique constraints) use user indexes if the indexes exist. If no user indexes can be used, system indexes remain nonfragmented and are moved to the dbspace where the database was created. To fragment a system index, create the fragmented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

Detaching an Index from a Table-Fragmentation Strategy

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

ADD Clause

In Dynamic Server, use the ADD clause to add another fragment to an existing fragmentation list.



Element	Purpose	Restrictions	Syntax
<i>existing_dbspace</i>	Name of a dbspace in an existing fragmentation list	The dbspace must exist at the time you execute the statement.	Identifier, p. 4-113
<i>expression</i>	Expression that defines the added fragment	The <i>expression</i> can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date and / or time functions, or aggregates are allowed in <i>expression</i> .	Condition, p. 4-5, and Expression, p. 4-33
<i>new_dbspace</i>	Name of dbspace to be added to the fragmentation scheme	The dbspace must exist at the time you execute the statement.	Identifier, p. 4-113

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 shown in the following example:

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

Adding Fragment Expressions

Adding a fragment expression to the fragmentation list in an expression-based distribution scheme can 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 fragmentation list, and a remainder fragment exists, the records in the remainder fragment are retrieved and re-evaluated. Some of these records may move to the new fragment. The remainder fragment always remains the last item in the fragment list.

IDS

DROP Clause

In Dynamic Server, use the DROP clause to drop an existing fragment from a fragmentation list.

DROP
Clause

[Back to ALTER FRAGMENT](#)
p. 2-8

→ DROP dbspace →

Element	Purpose	Restrictions	Syntax
<i>dbspace</i>	Name of the dbspace that contains the dropped fragment	The dbspace must exist at the time you execute the statement.	Identifier, p. 4-113

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 2-10](#) to increase your space, and retry the procedure.

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

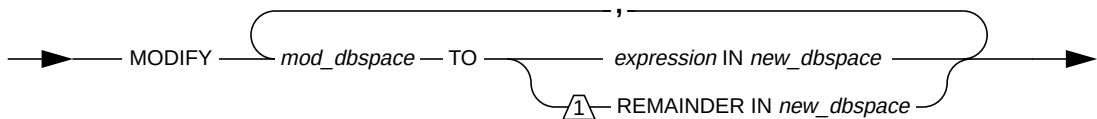
```
ALTER FRAGMENT ON INDEX cust_indx DROP dbsp2;
```

```
ALTER FRAGMENT ON TABLE customer DROP dbsp1;
```

IDS

MODIFY Clause

In Dynamic Server, 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.

MODIFY
ClauseBack to ALTER FRAGMENT
p. 2-8

Element	Purpose	Restrictions	Syntax
<i>expression</i>	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 and / or time functions, or aggregates are allowed in <i>expression</i> .	Condition, p. 4-5, and Expression, p. 4-33
<i>mod_dspace</i>	Modified dbspace	The dbspace must exist when you execute the statement.	Identifier, p. 4-113
<i>new_dspace</i>	Dspace that contains the modified information	The dbspace must exist when you execute the statement.	Identifier, p. 4-113

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 and the dbspace name.

What Happens to Indexes?

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

References

Related statements: CREATE TABLE, CREATE INDEX, and ALTER TABLE. Also see the Condition, Data Type, Expression, and Identifier segments.

For a discussion of fragmentation strategy, refer to the [*Informix Guide to Database Design and Implementation*](#).

For information on how to maximize performance when you make fragment modifications, see your [*Performance Guide*](#).

+

IDS

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.

You can use this statement only with Dynamic Server.

Syntax



Element	Purpose	Restrictions	Syntax
<i>index</i>	Name of the index to alter	The index must exist.	Database Object Name, p. 4-25

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 the index name belongs, the database server cannot execute the ALTER INDEX statement with the TO CLUSTER option; it returns an error unless lock mode is set to WAIT. (When lock mode is set to WAIT, the database server retries the ALTER INDEX statement.)

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

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

Related statements: CREATE INDEX and CREATE TABLE

For a discussion of the performance implications of clustered indexes, see your [Performance Guide](#).

+

ALTER TABLE

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

Syntax

ALTER TABLE *table* | *synonym* Alter Table Options p. [2-39](#)

Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of the table to alter	The table must exist.	Database Object Name, p. 4-25
<i>synonym</i>	Name of the synonym for the table to alter	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25

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.

You cannot alter a temporary table.

To add a referential constraint, you must have the DBA or References privilege on either the referenced columns or the referenced table.

When you add a constraint of any type, the name of the constraint must be unique in the database.

ANSI

In an ANSI-compliant database, when you add a constraint of any type, the *owner.name* combination (the combination of the owner name and constraint name) must be unique in the database. ♦

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.

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

Restrictions for Violations and Diagnostics Tables

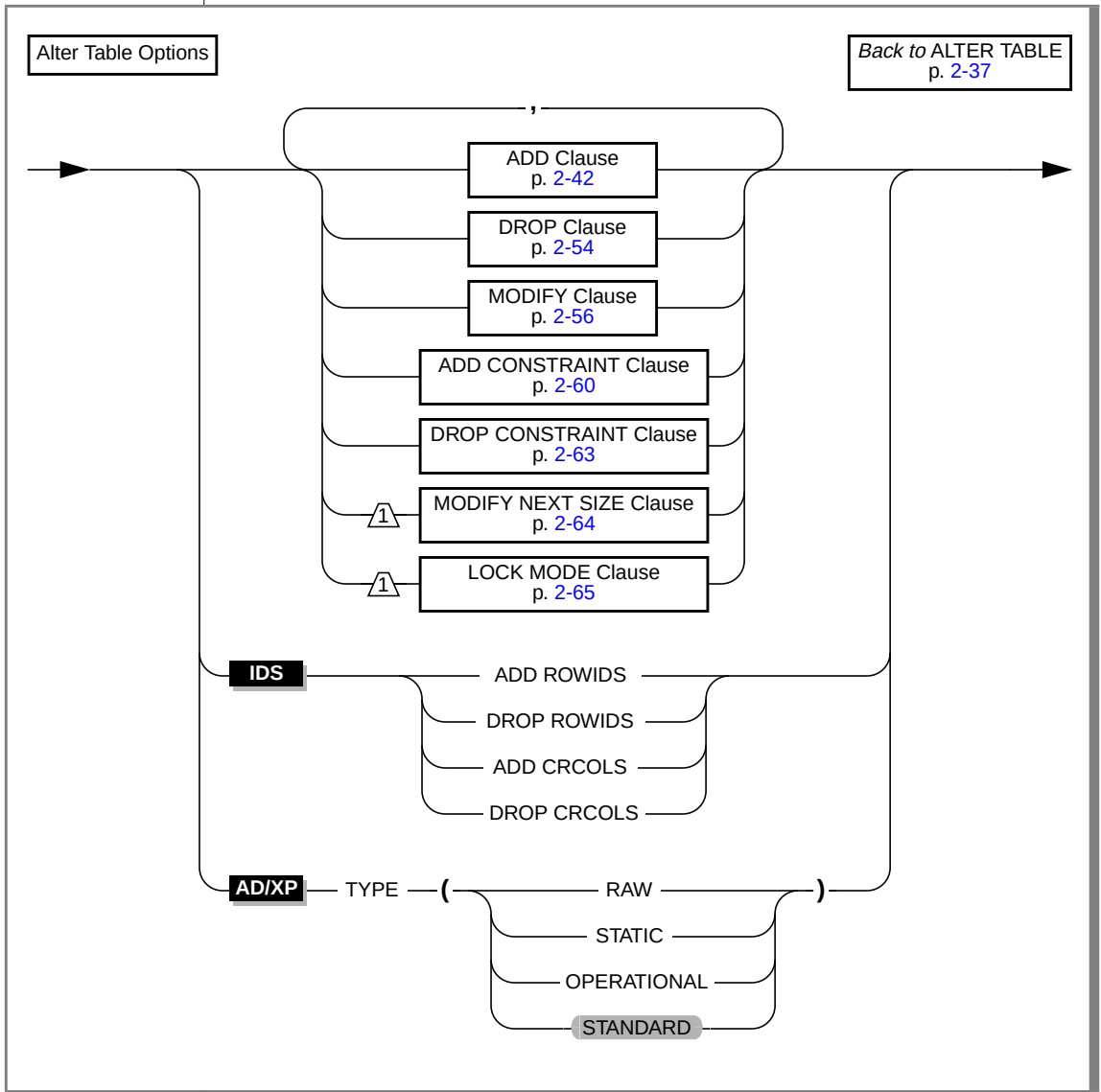
Keep the following considerations in mind when you use the ALTER TABLE statement in connection with violations and diagnostics tables:

- 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.

For further information on violations and diagnostics tables, see the START VIOLATIONS TABLE statement on [page 2-594](#).

Alter Table Options

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



IDS



IDS

IDS

ADD ROWIDS Clause

In Dynamic Server, fragmented tables do not contain the hidden rowid column by default. You use the ADD ROWIDS clause to add a new column called **rowid** to use with fragmented tables. The database server assigns a unique number to each row that remains stable for the life of the row. The database server creates an index that it uses to find the physical location of the row. The ADD ROWIDS clause cannot be used with other ALTER TABLE commands. After you add the rowid column, each row contains an additional 4 bytes to store the rowid value.

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

For additional information about the rowid column, refer to your [Administrator's Guide](#).

DROP ROWIDS Clause

In Dynamic Server, use the DROP ROWIDS clause to drop a rowid column only if you created it with the CREATE TABLE or ALTER FRAGMENT statements on fragmented tables. You cannot drop the rowid columns of a nonfragmented table. The DROP ROWIDS clause cannot be used with any other ALTER TABLE commands.

ADD CRCOLS and DROP CRCOLS Clauses

In Dynamic Server, use the ADD CRCOLS clause to create the shadow columns, **cdrserver** and **cdrtime**, that Enterprise Replication uses for conflict resolution. You must add these columns before you can use time-stamp or stored-procedure conflict resolution.

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

For more information, refer to [“WITH CRCOLS Keywords” on page 2-178](#) and to the [Guide to Informix Enterprise Replication](#).

TYPE Options

In Dynamic Server with AD and XP Options, the TYPE option is the only option allowed on a STATIC or RAW table.

Use the TYPE option to change the type of a permanent table to another permanent table type:

- RAW, a non-logging table that uses light appends
- STATIC, a non-logging table that can contain index and referential constraints
- OPERATIONAL, a logging table that uses light appends and is not restorable from archives
- STANDARD (default type of permanent table), a logging table that allows rollback, recovery, and restoration from archives

Restrictions on the TYPE Options

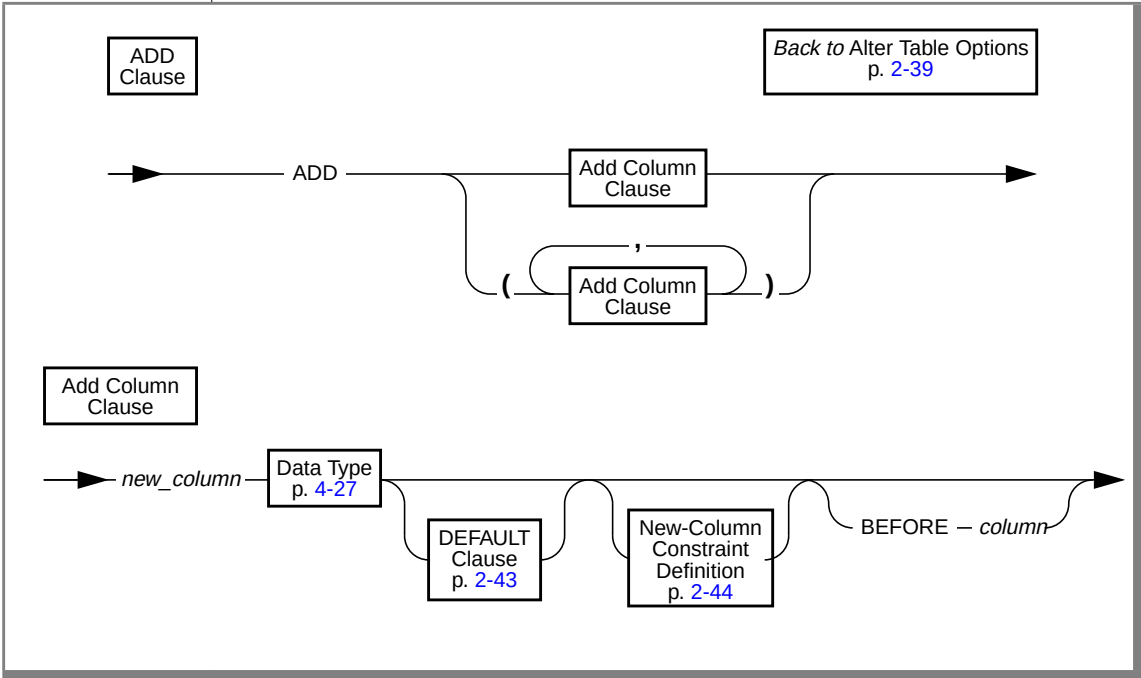
The type options have the following restrictions:

- You cannot change a table from a permanent type to a temporary type.
- You cannot change a TEMP or SCRATCH temporary type to any other type.
- You cannot change the table type if the table has a dependent GK index.
- You must perform a level-0 archive before a table can be altered to STANDARD type from any other type.

For a description of the four table types of permanent tables, refer to the [*Informix Guide to SQL: Reference*](#).

ADD Clause

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



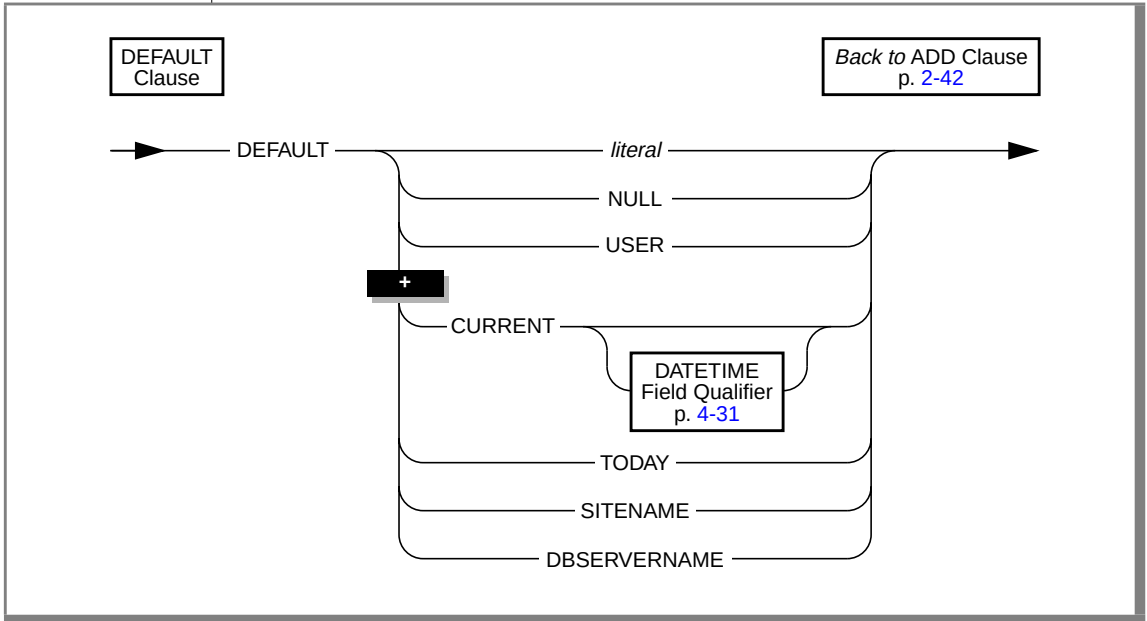
Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of the column before which the new column is to be placed	The column must already exist in the table.	Identifier, p. 4-113
<i>new_column</i>	Name of the column that you are adding	You cannot add a SERIAL column if the table contains data.	Identifier, p. 4-113

The following restrictions apply to the ADD clause:

- You cannot add a SERIAL column to a table if the table contains data.
- In Dynamic Server with AD and XP Options, you cannot add a column to a table that has a bit-mapped index. ♦

DEFAULT Clause

The default value is inserted into the column when an explicit value is not specified.

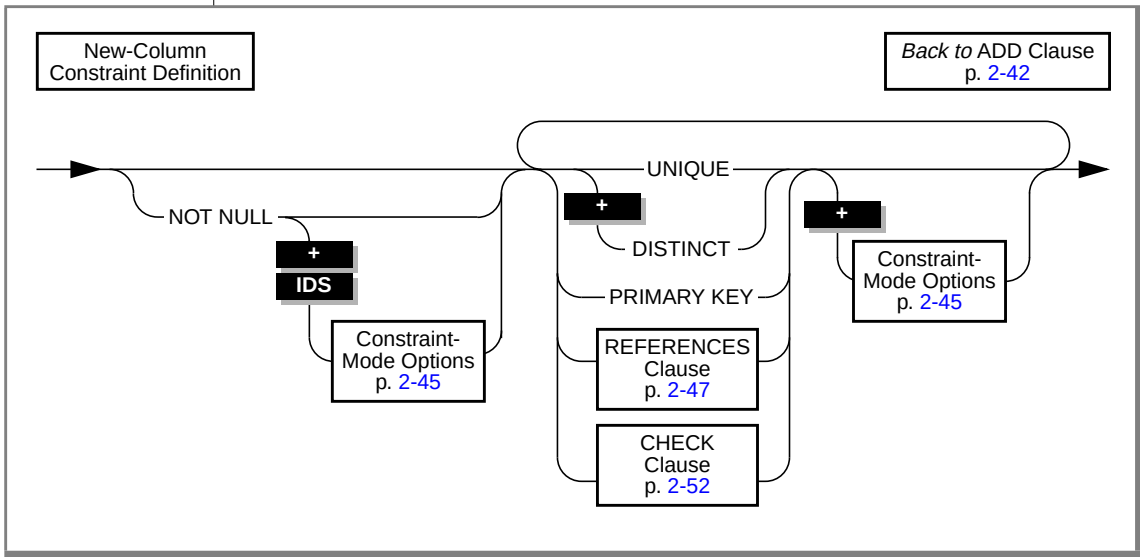


Element	Purpose	Restrictions	Syntax
<i>literal</i>	Literal term that defines alpha or numeric constant characters to use as the default value for the column	Term must be appropriate type for the column. See “ Literal Terms as Default Values ” on page 2-161.	Expression, p. 4-33

You cannot place a default on SERIAL columns.

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

For more information about the options of the **DEFAULT** clause, refer to the discussion of the **DEFAULT** clause for **CREATE TABLE**, page [2-160](#).

New-Column Constraint Definition

You cannot specify a primary-key or unique 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 primary-key or unique constraints on existing columns:

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

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

Using Not-Null Constraints with ADD

If a table contains data, when you add a column with a not-null constraint you must also include a DEFAULT clause. If the table is empty, no additional restrictions exist; that is, you can add a column and apply only the not-null constraint.

The following statement is valid whether or not the table contains data:

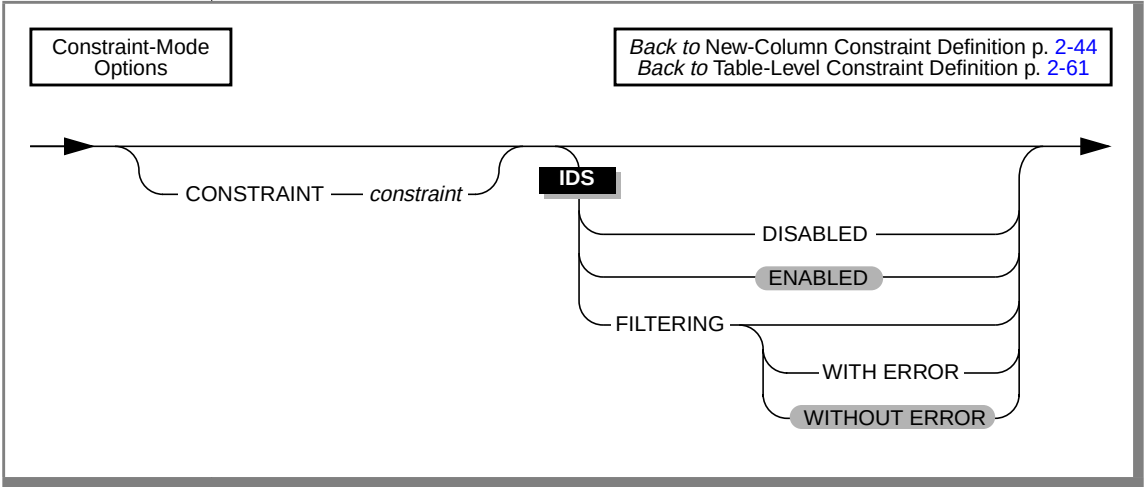
```
ALTER TABLE items
  ADD (item_weight DECIMAL(6,2)
      DEFAULT 2.0 NOT NULL
      BEFORE total_price)
```

Constraint Mode Options

In Dynamic Server, use the constraint mode options to assign a name to a constraint and to set the mode of the constraint to disabled, enabled, or filtering. ♦

AD/XP

In Dynamic Server with AD and XP Options, use the constraint mode options to assign a name to a constraint. ♦

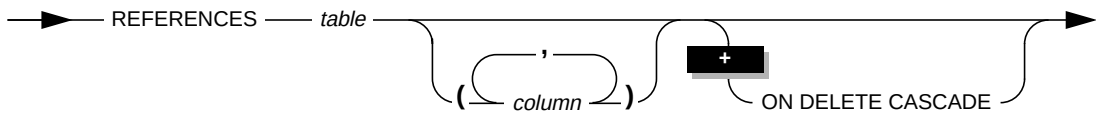


Element	Purpose	Restrictions	Syntax
<i>constraint</i>	Name assigned to the constraint	None.	Identifier, p. 4-113

For more information about constraint-mode options, refer to “[Constraint-Mode Options](#)” on page 2-172.

REFERENCES Clause

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

REFERENCES
Clause
[Back to New-Column Constraint Definition p. 2-44](#)
[Back to Table-Level Constraint Definition p. 2-61](#)


Element	Purpose	Restrictions	Syntax
<i>column</i>	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.	See “Restrictions on the Column Variable in the REFERENCES Clause” on page 2-48 .	Identifier, p. 4-113
<i>table</i>	Name of the referenced table	The referenced table can be the same table as the referencing table, or it can be a different table. The referenced and referencing tables must reside in the same database.	Database Object Name, p. 4-25

Restrictions on the Column Variable in the REFERENCES Clause

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

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

The following restrictions apply to the number of columns that you can specify in the *column* 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.
- You can specify a maximum of 16 column names. The total length of all the columns cannot exceed 255 bytes.

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 primary-key or unique 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 data type. When you add a column that references a SERIAL column, the column that you are adding must be an INTEGER column.

Using the ON DELETE CASCADE Clause

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 decide whether you want the rows in the child table deleted with the ON DELETE CASCADE clause. With the ON DELETE CASCADE clause (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?” on page 2-51](#).

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, the database 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 clause 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.

Restrictions on Cascading Deletes

Cascading deletes can be used for most deletes. One 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.

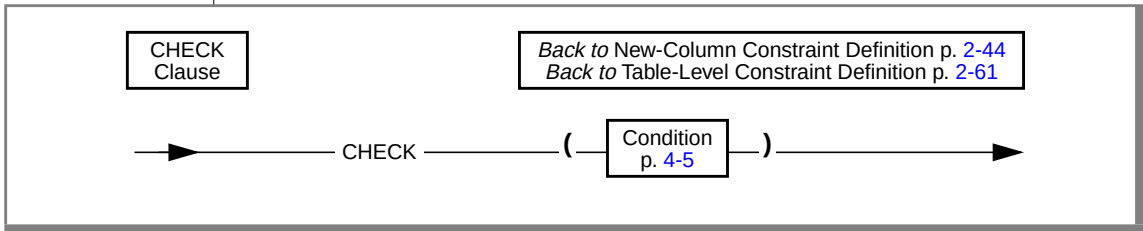
In addition, if a table has a trigger with a DELETE trigger event, you cannot define a cascading-delete referential constraint on that table. You receive an error when you attempt to add a referential constraint that specifies ON DELETE CASCADE to a table that has a delete trigger.

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).

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)
```

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:

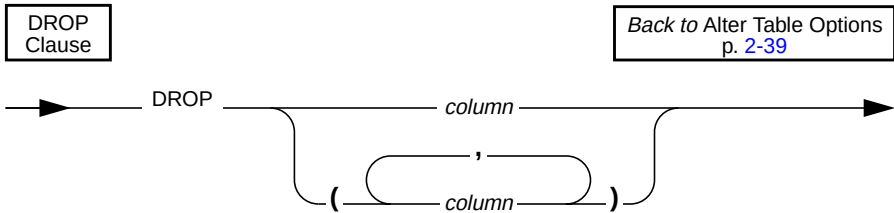
```
ALTER TABLE items  
ADD (item_weight DECIMAL(6,2) NOT NULL  
BEFORE total_price)
```

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)
```

DROP Clause

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



Element	Purpose	Restrictions	Syntax
column	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. If the column is the last column in the table, it cannot be dropped.	Identifier, p. 4-113

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

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

In Dynamic Server with AD and XP Options, you cannot use the DROP clause if the table has a dependent GK index. ♦

AD/XP

How Dropping a Column Affects Constraints

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

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

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

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

IDS***How Dropping a Column Affects Triggers***

In Dynamic Server, 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. For more information on triggering columns in an UPDATE trigger, see the CREATE TRIGGER statement on [page 2-198](#).

How Dropping a Column Affects Views

When you alter a table by dropping a column, view 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.

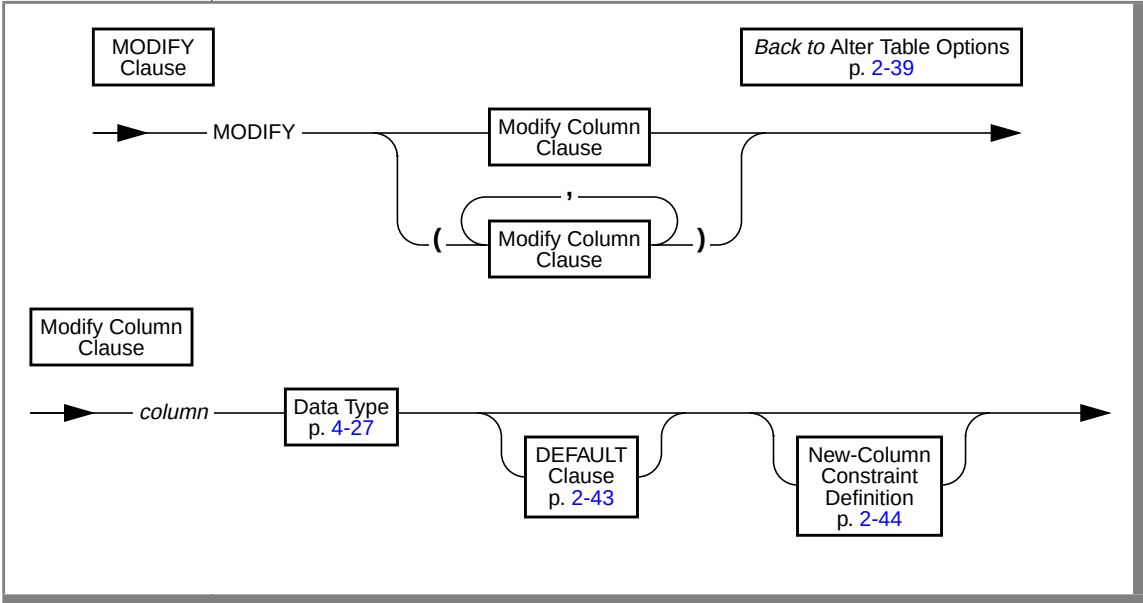
When you drop a column and add a new column with the same name, the changed order of columns does not drop views. Views based on the modified table continue to work; they retain their original sequence of columns.

AD/XP***How Dropping a Column Affects a Generalized-Key Index***

In Dynamic Server with AD and XP Options, if you drop a column from a table that has a dependent GK index, all GK indexes on the table that refer to the dropped column are dropped. Any GK indexes on other tables that refer to the dropped column are also dropped.

MODIFY Clause

Use the MODIFY clause to change the data type 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
<i>column</i>	Name of the column that you want to modify	The column must already exist in the table.	Identifier, p. 4-113

AD/XP

In Dynamic Server with AD and XP Options, you cannot use the **MODIFY** clause if the table has a dependent GK index. ♦

When you modify a column, *all* attributes previously associated with that column (that is, default value, single-column check constraint, or referential constraint) are dropped. When you want certain attributes of the column to remain, such as PRIMARY KEY, you must re-specify those attributes. For example, if you are changing the data type of an existing column, **quantity**, to SMALLINT, 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 primary-key or unique constraint, all referential constraints that reference the multiple columns also are dropped.

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

In Dynamic Server, if you modify a column that appears in the triggering column list of an UPDATE trigger, the trigger is unchanged. ♦

Altering BYTE and TEXT Columns

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



Altering the Next Serial Number

You can use the **MODIFY** clause to reset the next value of a **SERIAL** column. You cannot set the next value below the current maximum value in the column because that action can cause the database server to generate duplicate numbers. 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 primary-key or unique constraint exists, however, conversion takes place only if it does not violate the constraint. If a data-type conversion would result in duplicate values (by changing **FLOAT** to **SMALLFLOAT**, for example, or by truncating **CHAR** values), the **ALTER TABLE** statement fails.

Modifying Tables for Null Values

You can modify an existing column that formerly permitted nulls to disallow nulls, provided that the column contains no null values. To do this, specify **MODIFY** with the same column name and data type and the **NOT NULL** keywords. 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.

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

In Dynamic Server, 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 database object mode of the constraint to the enabled mode.
When you issue this statement, existing rows in the target table that violate the constraint are duplicated in the violations table; however, you receive an integrity-violation error message, and the constraint remains disabled.
4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table.
You might need to join the violations and diagnostics tables to get all the necessary information.
5. Take corrective action on the rows in the target table that violate the constraint.
6. After you fix all the nonconforming rows in the target table, issue the SET statement again to switch the disabled constraint to the enabled mode.

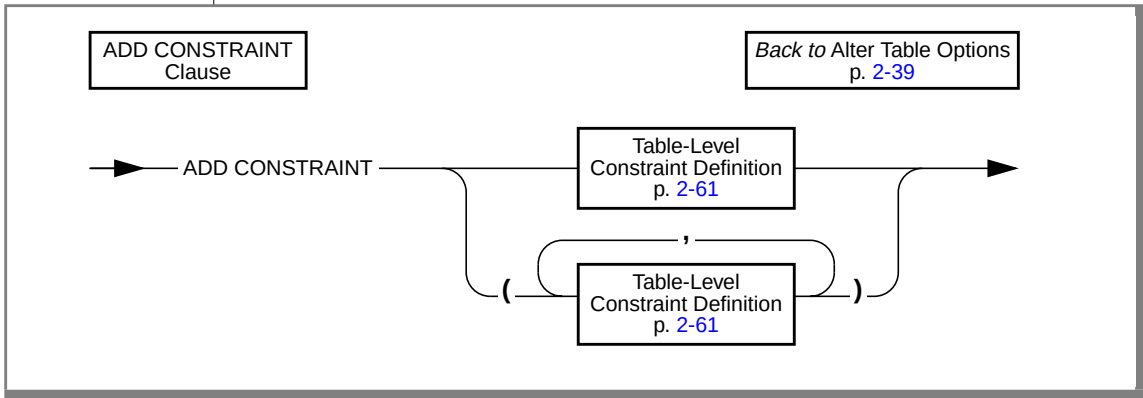
This time the constraint is enabled, and no integrity-violation error message is returned because all rows in the target table now satisfy the new constraint.

How Modifying a Column Affects a Generalized-Key Index

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

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

ADD CONSTRAINT Clause



Use `ADD CONSTRAINT` with the `ALTER TABLE` statement 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)
```

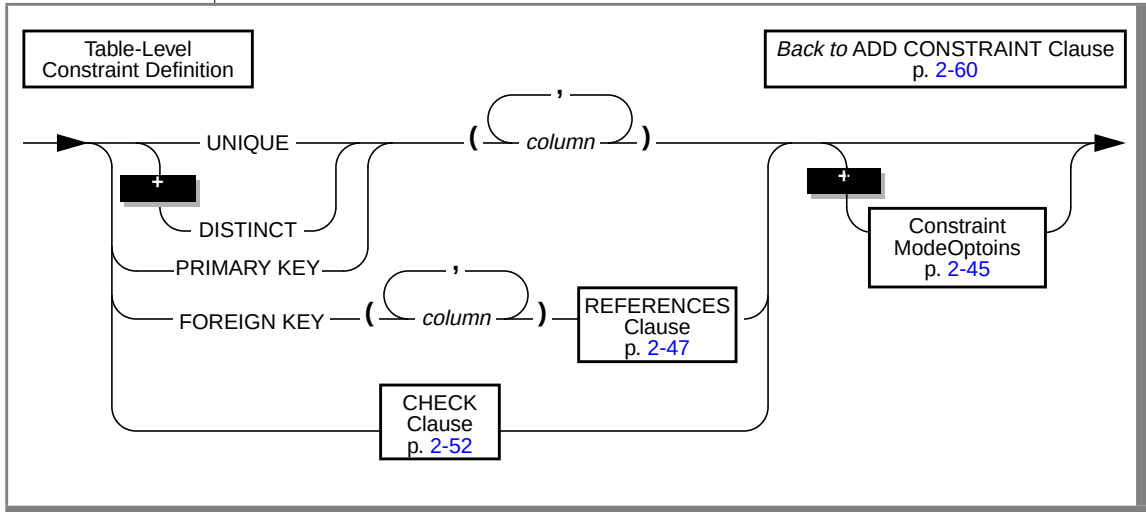
To name the constraint, change the preceding statement, as shown in the following example:

```
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, see the *Informix Guide to SQL: Reference*.

Table-Level Constraint Definition

Use the Table-Level Constraint Definitions option to add a table-level constraint.



Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of the column or columns on which the constraint is placed	The maximum number of columns is 16, and the total length of the list of columns cannot exceed 255 bytes.	Identifier, p. 4-113

You can define a table-level constraint on one column or a set of columns. A constraint that involves multiple columns can include no more than 16 column names. The total length of the list of columns cannot exceed 255 bytes.

You can assign a name to the constraint and set its mode by means of the Constraint Mode Definitions option. For more information about the individual constraint modes, see [“Constraint Mode Options” on page 2-45](#).

The name of the constraint must be unique within the database.

If you are using an ANSI-compliant database, the *owner.name* combination (the combination of the owner name and constraint name) must be unique in the database. ♦

Adding a Primary-Key or Unique Constraint

When you place a primary key or unique constraint on a column or set of columns, those columns can contain only unique values.

When you place a primary-key or unique constraint on a column or set of columns, and a unique index already exists on that column or set of columns, the constraint shares the index. However, if the existing index allows duplicates, the database server returns an error. You must drop the existing index before adding the constraint.

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

Adding a Referential Constraint

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

Privileges Required for Adding Constraints

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

Recovery from Constraint Violations

In Dynamic Server, 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 2-59.

DROP CONSTRAINT Clause

DROP CONSTRAINT
Clause

[Back to Alter Table Options](#)
p. 2-39



Element	Purpose	Restrictions	Syntax
<i>constraint</i>	Name of the constraint that you want to drop	The constraint must exist.	Database Object Name, p. 4-25

Use the DROP CONSTRAINT clause to drop a named constraint.

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:

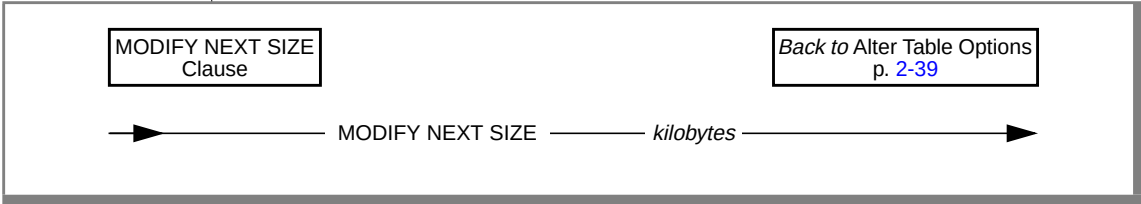
```
ALTER TABLE manufact DROP CONSTRAINT con_name
```

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

MODIFY NEXT SIZE Clause



Element	Purpose	Restrictions	Syntax
<i>kilobytes</i>	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. 4-33

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
```

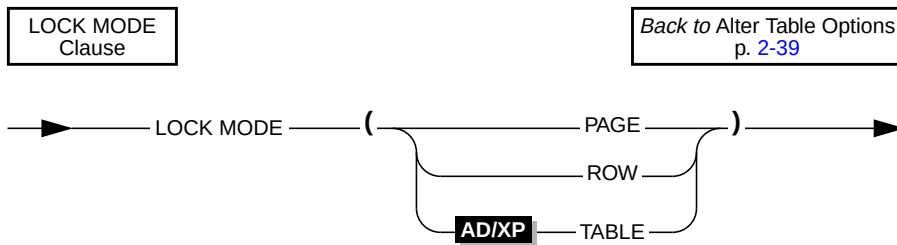
The size of existing extents is not changed. You cannot change the size of existing extents without unloading all of the data.

Changing the Size of Existing Extents

To change the size of existing extents, you must unload all of the data, modify the extent and next-extent sizes in the CREATE TABLE statement of the database schema, re-create the database, and reload the data. For information about revising extent sizes, see your [Administrator's Guide](#).

LOCK MODE Clause

Use the LOCK MODE keywords to change the locking mode of a table. The default lock mode is 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. You can also exceed the maximum number of locks available, depending on the configuration of your operating 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.

AD/XP

In Dynamic Server with AD and XP Options, table locking provides a lock on an entire table. This type of lock reduces update concurrency in comparison to row and page locks. Multiple read-only transactions can still access the table. A table lock reduces the lock-management overhead for the table. ♦

References

Related statements: CREATE TABLE, DROP TABLE, LOCK TABLE, and SET DATABASE OBJECT MODE

For discussions of data-integrity constraints and the ON DELETE CASCADE clause, see the [Informix Guide to SQL: Tutorial](#).

For a discussion of database and table creation, see the [Informix Guide to Database Design and Implementation](#).

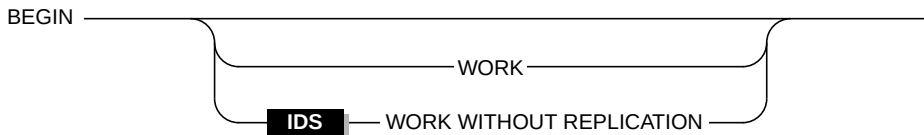
For information on how to maximize performance when you make table modifications, see your [Performance Guide](#).

+

BEGIN WORK

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

Syntax



Usage

Each row that an `UPDATE`, `DELETE`, or `INSERT` statement affects during a transaction is locked and remains locked throughout the transaction. A transaction that contains many such statements or that contains statements affecting many rows can exceed the limits that your operating system or the database server configuration imposes on the maximum number of simultaneous locks. If no other user is accessing the table, you can avoid locking limits and reduce locking overhead by locking the table with the `LOCK TABLE` statement after you begin the transaction. Like other locks, this table lock is released when the transaction terminates. The example of a transaction on [page 2-69](#) includes a `LOCK TABLE` statement.

Important: 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.



E/C

In `ESQL/C`, 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. ♦

ANSI

WORK Keyword

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

```
BEGIN;
```

```
BEGIN WORK;
```

BEGIN WORK and ANSI-Compliant Databases

In an ANSI-compliant database, you do not need the BEGIN WORK statement 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.

IDS

BEGIN WORK WITHOUT REPLICATION

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

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

For more information about data replication, see the [Guide to Informix Enterprise Replication](#).

Example of BEGIN WORK

The following code fragment shows how you might place statements within a transaction. The transaction is made up of the statements that occur between the BEGIN WORK and COMMIT WORK statements. The transaction locks the **stock** table (LOCK TABLE), updates rows in the **stock** table (UPDATE), deletes rows from the **stock** table (DELETE), and inserts a row into the **manufact** table (INSERT). The database server must perform this sequence of operations either completely or not at all. The database server guarantees that all the statements are completely and perfectly committed to disk, or the database is restored to the same state as before the transaction began.

```
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;
```

References

Related statements: COMMIT WORK, ROLLBACK WORK

For discussions of transactions and locking, see the [Informix Guide to SQL: Tutorial](#).

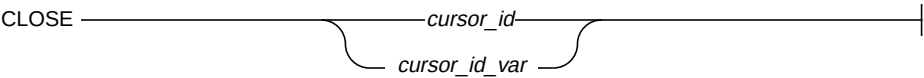
E/C

CLOSE

Use the CLOSE statement when you no longer need to refer to the rows that a select or procedure cursor produced or when you want to flush and close an insert cursor.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of the cursor to close	The DECLARE statement must have previously declared the cursor.	Identifier, p. 4-113
<i>cursor_id_var</i>	Host variable that holds the value of <i>cursor_id</i>	Host variable must be a character data type. The cursor must be declared. In ANSI-compliant databases, before you can close a cursor, the cursor must be open.	Name must conform to language-specific rules for variable names.

Usage

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

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

ANSI

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

Closing a Select or Procedure Cursor

When a cursor identifier is associated with a `SELECT` or `EXECUTE PROCEDURE` statement, closing the cursor 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 procedure cursor, you cannot execute a `FETCH` statement that names that cursor until you have reopened it.

Closing an Insert Cursor

When a cursor identifier is associated with an `INSERT` statement, the `CLOSE` statement writes any remaining buffered rows into the database. The number of rows that were successfully inserted into the database is returned in the third element of the `sqlerrd` array, `sqlca.sqlerrd[2]`, in the `sqlca` structure. For information on using `SQLERRD` to count the total number of rows that were inserted, see the `PUT` statement on [page 2-417](#).

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

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



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

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

Using End of Transaction to Close a Cursor

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

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

For how to use insert cursors and the WITH HOLD clause, see the DECLARE statement on [page 2-241](#).

ANSI

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

References

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

For a discussion of cursors, see the [Informix Guide to SQL: Tutorial](#).

+

CLOSE DATABASE

Use the CLOSE DATABASE statement to close the current database.

Syntax

```
CLOSE DATABASE _____|
```

Usage

Following the CLOSE DATABASE statement, the only legal SQL statements are CREATE DATABASE, DATABASE, and DROP DATABASE. 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
```

E/C

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

CLOSE DATABASE

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 re-declare any cursors that you want to use. ♦

References

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

COMMIT WORK

Use the COMMIT WORK statement to commit all modifications made to the database from the beginning of a transaction. This statement informs the database server that you reached the end of a series of statements that must succeed as a single unit. The database server takes the required steps to make sure that all modifications made by the transaction are completed correctly and committed to disk.

Syntax

```
COMMIT _____|
                |
                | WORK
```

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.

In ESQL/C, the COMMIT WORK statement closes all open cursors except those declared with hold. ♦

The following example shows a transaction bounded by BEGIN WORK and COMMIT WORK statements. In this example, the user first deletes the row from the **call_type** table where the value of the **call_code** column is 0. The user then inserts a new row in the **call_type** table where the value of the **call_code** column is S. The database server guarantees that both operations succeed or else neither succeeds.

```
BEGIN WORK;
  DELETE FROM call_type WHERE call_code = '0';
  INSERT INTO call_type VALUES ('S', 'order status');
COMMIT WORK;
```

E/C

ANSI

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

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.

WORK Keyword

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

```
COMMIT;
```

```
COMMIT WORK;
```

References

Related statements: BEGIN WORK, ROLLBACK WORK, and DECLARE

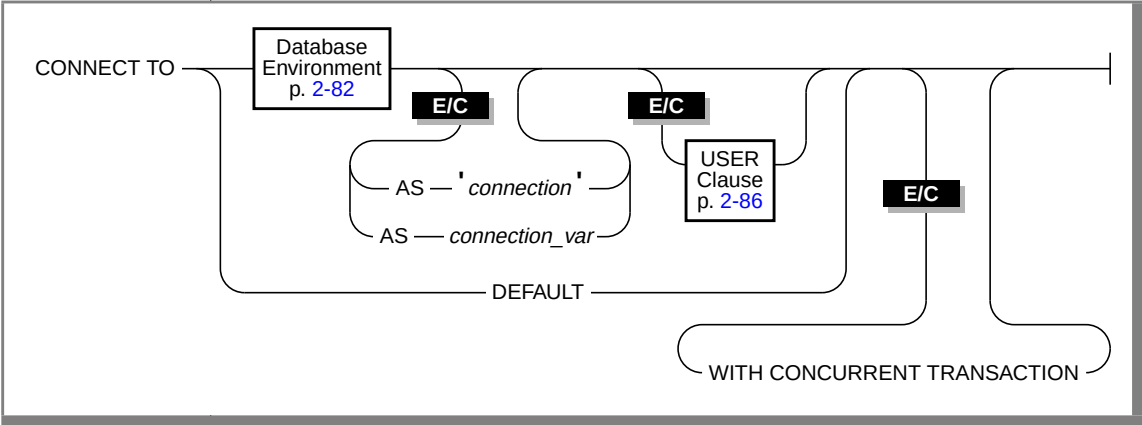
For a discussion of concepts related to transactions, see the [Informix Guide to SQL: Tutorial](#).

+

CONNECT

Use the CONNECT statement to connect to a database environment.

Syntax



Element	Purpose	Restrictions	Syntax
connection	Quoted string that assigns a name to the connection	Each connection name must be unique.	Quoted String, p. 4-157
connection_var	Host variable that holds the value of connection	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.

UNIX

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.

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

WIN NT

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

Only one connection is current at any time; other connections are dormant. The application cannot interact with a database through a dormant connection. When an application establishes a new connection, that connection becomes current, and the previous current transaction becomes dormant. You can make a dormant connection current with the SET CONNECTION statement. For more information, see [“SET CONNECTION” on page 2-505](#).

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 2-86](#).

Connection Identifiers

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

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

The value of a 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 2-506.](#))

DEFAULT Option

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

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

The Implicit Connection with DATABASE Statements

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

- ★ DATABASE
- ★ CREATE DATABASE
- ★ DROP DATABASE

If one of these database statements is the first SQL statement in an application, the statement establishes a connection to a 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 2-84](#).

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 CONCURRENT TRANSACTION clause:

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

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

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

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

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

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

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

```

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

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

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

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

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

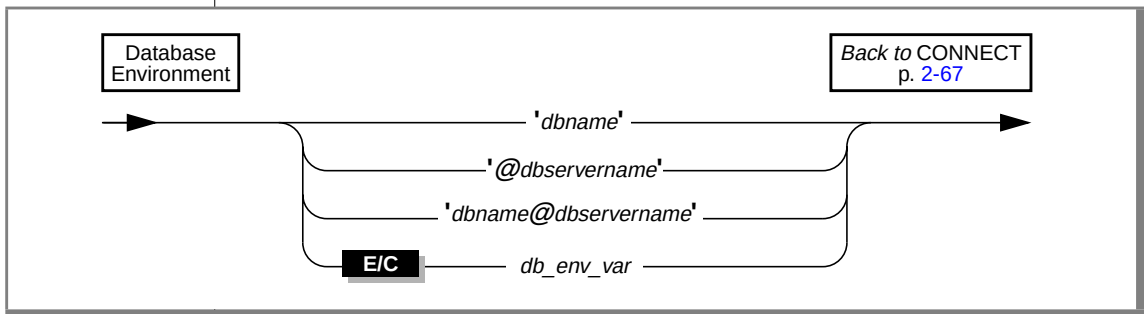
EXEC SQL disconnect all;
}

```



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

Database Environment



Element	Purpose	Restrictions	Syntax
<i>db_env_var</i>	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.
<i>dbname</i>	Name of the database to which a connection is made	The specified database must already exist.	Identifier, p. 4-113
<i>dbservername</i>	Name of the database server to which a connection is made	The specified database server must exist. You cannot put a space between the @ symbol and <i>dbservername</i> .	Identifier, p. 4-113

Using Quote Marks in the Database Environment

If the **DELIMIDENT** environment variable is set, the quote marks in the database environment must be single. If the **DELIMIDENT** environment variable is not set, surrounding quotes can be single or double.

Restrictions on the dbservername Parameter

When the *dbservername* parameter appears in the specification of a database environment, you must observe the following restrictions.

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

On Windows NT, the database server that you specify in *dbservername* must match the name of a database server in the **sqlhosts** subkey in the registry. Informix recommends that you use the **setnet32** utility to update the registry. ♦

UNIX

WIN NT

Specifying the Database Environment

Using the options 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 (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 a connection to the default server or to another database server in the DBPATH variable. It also locates and opens the named database. The same is true of the `db_env_var` option if it specifies only a database name. For the order in which an application connects to different database servers to locate a database, see “Locating the Database”.

Locating the Database

How a database is located and opened depends 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.

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, 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 before the **DBPATH** value.

UNIX

On UNIX, set the **INFORMIXSERVER** and **DBPATH** environment variables as the following example shows:

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

**WIN NT**

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

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



The resulting **DBPATH** that your application uses is shown in the following example:

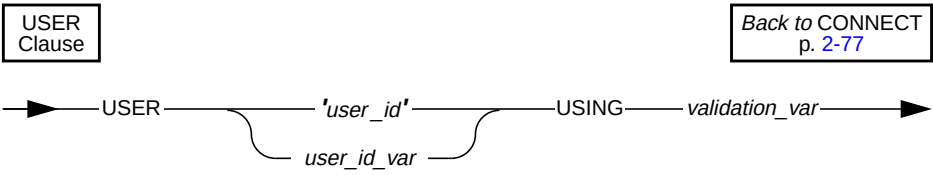
```
//srvA://srvB://srvC
```

The application first establishes a connection to the database server specified by **INFORMIXSERVER**. The database server 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, that server would be **srvB**.

USER Clause

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.



Element	Purpose	Restrictions	Syntax
validation_var	Host variable that holds the valid password for the login name specified in user_id or user_id_var	Variable must be a fixed-length character data type. The password stored in this variable must be a valid password. For additional restrictions see “Restrictions on the Validation Variable Parameter” on page 2-87.	Variable name must conform to language-specific rules for variable names.
user_id	Quoted string that is a valid login name for the application	The specified login name must be a valid login name. For additional restrictions see “Restrictions on the User Identifier Parameter” on page 2-87.	Quoted String, p. 4-157
user_id_var	Host variable that holds the value of user_id	Variable must be a fixed-length character data type. The login name stored in this variable is subject to the same restrictions as user_id.	Variable name must conform to language-specific rules for variable names.

UNIX

Restrictions on the Validation Variable Parameter

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

WIN NT

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

UNIX

Restrictions on the User Identifier Parameter

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

WIN NT

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

Rejection of the Connection

The connection is rejected if the following conditions occur:

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

E/C

X/O

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

Use of the Default User ID

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

References

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

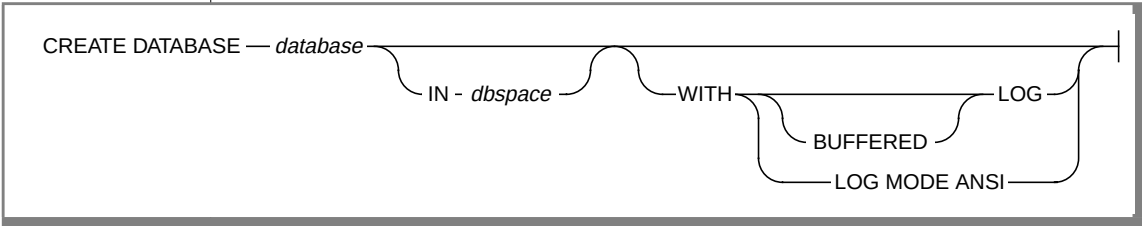
For more information about **sqlhosts**, refer to your [Administrator's Guide](#).

+

CREATE DATABASE

Use the CREATE DATABASE statement to create a new database.

Syntax



Element	Purpose	Restrictions	Syntax
<i>database</i>	Name of the database to create	The database name must be unique on the server.	Database Name, p. 4-22
<i>dbspace</i>	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. 4-113

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 describe the structure of the database.

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 how to grant database privileges, see the GRANT statement on page 2-342.

E/C

In ESQ/C, the CREATE DATABASE statement cannot appear in a multistatement PREPARE operation. ♦

If you do not specify the dbspace, the database server creates the system catalog tables in the **root** dbspace. The following statement creates the **vehicles** database in the **root** dbspace:

```
CREATE DATABASE vehicles
```

The following statement creates the **vehicles** database in the **research** dbspace:

```
CREATE DATABASE vehicles IN research
```

Logging Options

The logging options of the CREATE DATABASE statement determine the type of logging that is done for the database.

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

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

If you are using Dynamic Server with AD and XP Options, the CREATE DATABASE statement always creates a database with unbuffered logging. The database server ignores any logging specifications included in a CREATE DATABASE statement. ♦

Designating Buffered Logging

The following example creates a database that uses a buffered log:

```
CREATE DATABASE vehicles WITH BUFFERED LOG
```

If you use a buffered log, you marginally enhance the performance of logging at the risk of not being able to re-create the last few transactions after a failure. (See the discussion of buffered logging in the [Informix Guide to SQL: Tutorial](#).)

ANSI

ANSI-Compliant Databases

When you use the LOG MODE ANSI option in the CREATE DATABASE statement, the database that you create is an ANSI-compliant database. ANSI-compliant databases are set apart from databases that are not ANSI-compliant by the following features:

- All statements are automatically contained in transactions. All databases use unbuffered logging.
- Owner-naming is enforced. You must use the owner name when you refer to each table, view, synonym, index, or constraint unless you are the owner.
- For databases, the default isolation level is repeatable read.
- Default privileges on objects differ from those in databases that are not ANSI-compliant. Users do not receive PUBLIC privilege to tables and synonyms by default.

Other slight differences exist between databases that are ANSI-compliant and those that are not. These differences are noted as appropriate with the related SQL statement. For a detailed discussion of the differences between ANSI-compliant databases and databases that are not ANSI-compliant, see the [Informix Guide to SQL: Reference](#).

Creating an ANSI-compliant database does not mean that you get 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 the [Informix Guide to SQL: Reference](#).

References

Related statements: CLOSE DATABASE, CONNECT, DATABASE, DROP DATABASE

For discussions of how to create a database and of ANSI-compliant databases, see the [Informix Guide to Database Design and Implementation](#).

+

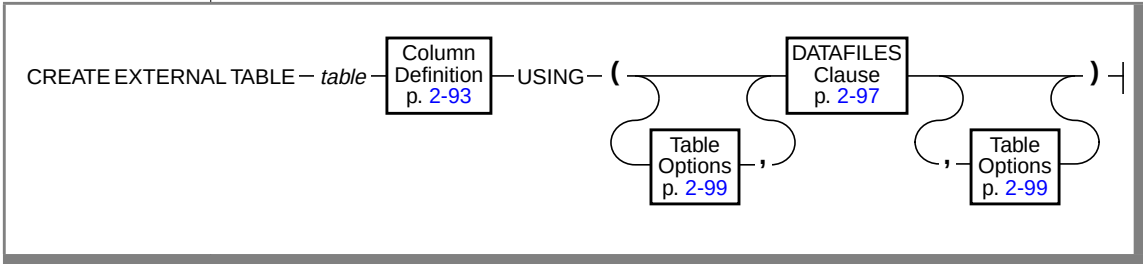
AD/XP

CREATE EXTERNAL TABLE

Use the CREATE EXTERNAL TABLE statement to define an external source that is not part of your database so you can use that external source to load and unload data for your database.

You can use this statement only with Dynamic Server with AD and XP Options.

Syntax

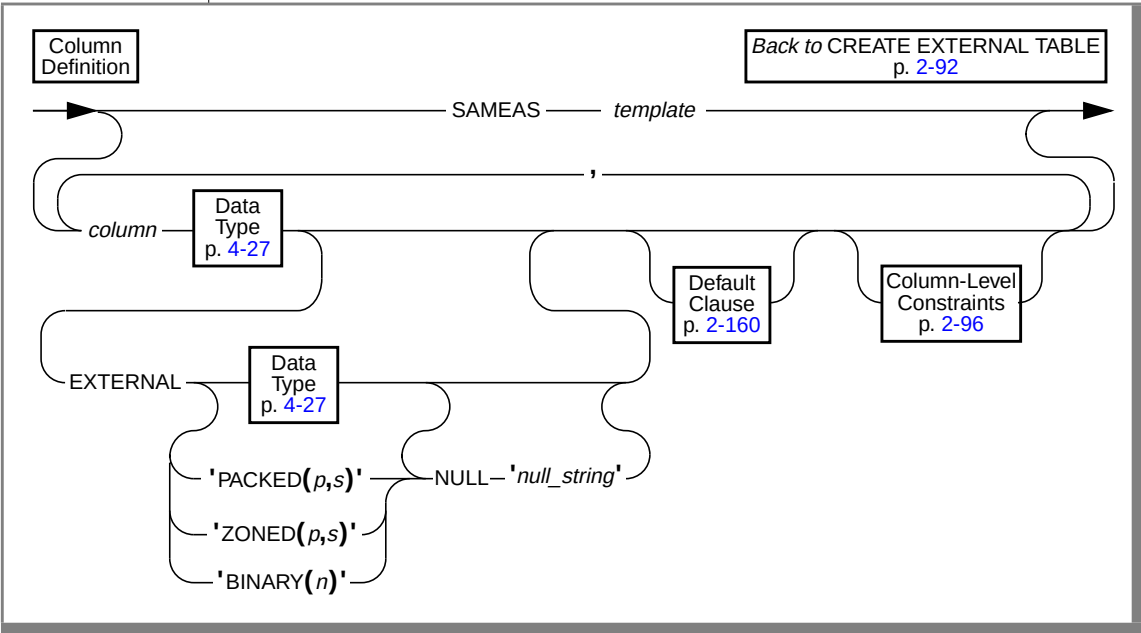


Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of the external table that describes the external data	The name must be different from any existing table, view, or synonym name in the current database.	Database Object Name, p. 4-25

Usage

After you create a table with the CREATE EXTERNAL TABLE statement, you can move data to and from the external source with an INSERT INTO...SELECT statement.

Column Definition



Element	Purpose	Restrictions	Syntax
column	One column name for each column of the external table	For each column, you must specify an Informix data type.	Identifier, p. 4-113
p	Precision (total number of digits)	For FIXED-format files only. If NULL values are allowed, you must include a null string.	Literal Number, p. 4-139
s	Scale (number of digits after the decimal point)	For FIXED-format files only. If NULL values are allowed, you must include a null string.	Literal Number, p. 4-139
n	Number of 8-bit bytes to represent the integer	For FIXED format binary integers; “big-endian” byte order. If NULL values are allowed, you must include a null string.	n=2 for 16-bit integers; n=4 for 32-bit integers

Element	Purpose	Restrictions	Syntax
<i>null_string</i>	Value that is to be interpreted as a null	For restriction information, refer to “Defining Null Values.”	Quoted String, p. 4-157
<i>template</i>	Name of a table with the exact column names and definitions for your external table	You cannot skip columns or put data of a different type into any column.	Database Object Name, p. 4-25

(2 of 2)

Using the SAMEAS Clause

When you create a table with the SAMEAS keyword, the column names from the original table are used in the new table. You cannot use indexes in the external table definition.

You cannot use the SAMEAS keyword for FIXED-format files. For files in FIXED format, you must declare the column name and the EXTERNAL item for each column to set the name and number of characters. You can use the keyword NULL to specify what string to interpret as a NULL value.

Using the EXTERNAL Keyword

Use the EXTERNAL keyword to specify data type for each column of your external table that has a data type different from the internal table. For example, you might have a VARCHAR column in the internal table that you want to map to a CHAR column in the external table.

The different data type can be an Informix data type or one of the types specified in the following section, “Additional Representations for Integers.” For FIXED-format files, the only data type allowed is CHAR.

Additional Representations for Integers

Besides valid Informix integer data types, you can specify packed decimal, zoned decimal, and IBM-format binary representation of integers.

For packed or zoned decimal, you specify precision (total number of digits in the number) and scale (number of digits that are to the right of the decimal point). Packed decimal representation can store two digits, or a digit and a sign, in each byte. Zoned decimal requires ($p + 1$) bytes to store p digits and the sign.

Big-Endian Format

The database server also supports two IBM-format binary representations of integers: BINARY(2) for 16-bit integer storage and BINARY(4) for 32-bit integer storage. The most significant byte of each number has the lowest address; that is, binary-format integers are stored big end first (“big-endian format”) in the manner of IBM and Motorola processors. Intel processors and some others store binary-format integers little end first, a storage method that the database server does not support for external data.

Defining Null Values

The packed decimal, zoned decimal, and binary data types do not have a natural null value, so you must define a value that can be interpreted as a null when the database server loads or unloads data from an external file. You can define the *null_string* as a number that will not be used in the set of numbers stored in the data file (for example, -9999.99). You can also define a bit pattern in the field as a hexadecimal pattern, such as 0xffff, that is to be interpreted as a null.

The database server uses the null representation for a fixed-format external table to both interpret values as the data is loaded into the database and to format null values into the appropriate data type when data is unloaded to an external table.

The following are examples of column definitions with null values for a fixed-format external table.

```
i smallint external "binary (2)" null "-32767"
li integer external "binary (4)" null "-99999"
d decimal (5,2) external "packed (5,2)" null "0xffffffff"
z decimal (4,2) external "zoned (4,2)" null "0x0f0f0f0f"
zl decimal (3,2) external "zoned (3,2)" null "-1.00"
```

If the packed decimal or zoned decimal is stored with all bits cleared to represent a null value, the *null_string* can be defined as “0x0”. The following rules apply to the value assigned to a *null_string*:

- The null representation must fit into the length of the external field.
- If a bit pattern is defined, the *null_string* is not case sensitive.
- If a bit pattern is defined, the *null_string* must begin with “0x”.

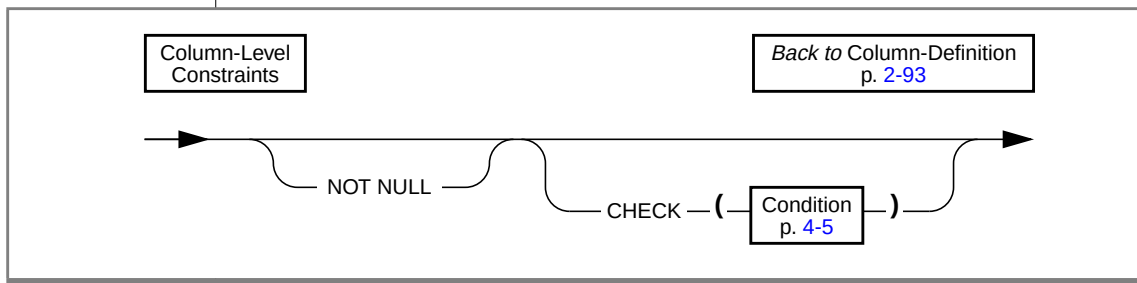


- For numeric fields, the left-most fields are assigned zeros by the database server if the bit pattern does not fill the entire field.
- If the null representation is not a bit pattern, the null value must be a valid number for that field.

Warning: *If a row that contains a null value is unloaded into an external table and the column that receives the null value has no null value defined, the database server inserts a zero into the column.*

Column-Level Constraints

Use column-level constraints to limit the type of data that is allowed in a column. Constraints at the column level are limited to a single column.



Using the Not-Null Constraint

If you do not indicate a default value for a column, the default is null *unless* you place a not-null constraint on the column. In that case, no default value exists for the column.

If you place a not-null constraint on a column (and no default value is specified), the data in the external table must have a value set for the column when loading through the external table. When no reject file exists and no value (or a null value) is encountered, the database server returns an error and the loading stops. When a reject file exists and a null value is encountered, the error is reported in the reject file and the load continues.

Using the CHECK Constraint

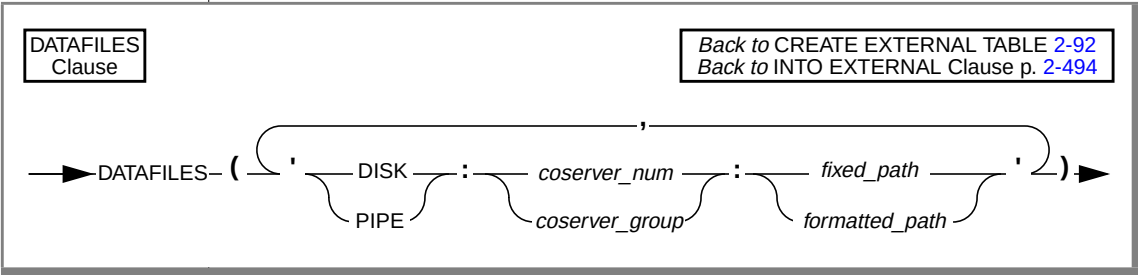
Check constraints allow you to designate conditions that must be met *before* data can be assigned to a column during an INSERT or UPDATE statement. When a reject file does not exist and a row evaluates to *false* for any check constraint defined on a table during an insert or update, the database server returns an error. When there is a reject file and a row evaluates to *false* for a check constraint defined on the table, the error is reported in the reject file and the statement continues to execute.

Check constraints are defined with *search conditions*. The search condition cannot contain subqueries; aggregates; host variables; the CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY functions; or stored procedure calls.

When you define a check constraint at the column level, the only column that the check constraint can check against is the column itself. In other words, the check constraint cannot depend upon values in other columns of the table.

DATAFILES Clause

The DATAFILES clause names the external files that are opened when you use external tables.



Element	Purpose	Restrictions	Syntax
coserver_num	Numeric ID of the coserver that contains the external data	The coserver must exist.	Literal Number, p. 4-139
coserver_group	Name of the coserver group that contains the external data	The coserver group must exist.	Identifier, p. 4-113

CREATE EXTERNAL TABLE

Element	Purpose	Restrictions	Syntax
<i>fixed_path</i>	Pathname for describing the input or output files in the external table definition	The specified path must exist.	The pathname must conform to the conventions of your operating system.
<i>formatted_path</i>	Formatted pathname that uses pattern-matching characters	The specified path must exist.	The pathname must conform to the conventions of your operating system.

(2 of 2)

You can use cogroup names and coserver numbers when you describe the input or output files for the external table definition. You can identify the DATAFILES either by coserver number or by cogroup name. A coserver number contains only digits. A cogroup name is a valid identifier that begins with a letter but otherwise contains any combination of letters, digits, and underscores.

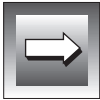
If you use only some of the available coservers for reading or writing files, you can designate these coservers as a cogroup using **onutil** and then use the cogroup name rather than explicitly naming each coserver and file separately. Whenever you use all coservers to manage external files, you can use the predefined *coserver_group*.

For examples of the DATAFILES clause, see the section, [“Examples” on page 2-104](#).

Using Formatting Characters

You can use a formatted pathname to designate a filename. If you use a formatted pathname, you can take advantage of the substitution characters %c, %n, and %r(first...last).

Formatting String	Significance
%c	Replaced with the number of the coserver that manages the file
%n	Replaced with the name of the node on which the coserver that manages the file resides
%r(first...last)	Names multiple files on a single coserver

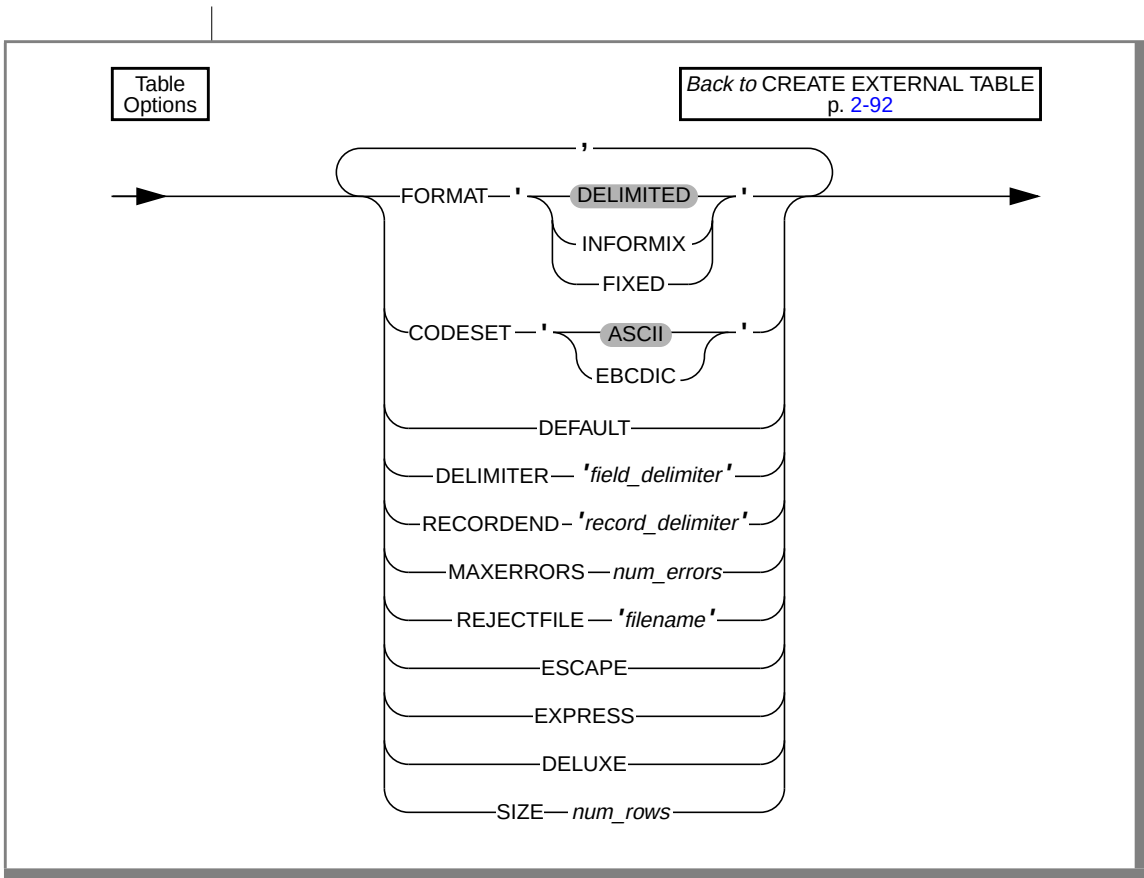


Important: The formatted pathname option does not support the %o formatting string.

Table Options

The optional table parameters include additional characteristics that define the table.

CREATE EXTERNAL TABLE



Element	Purpose	Restrictions	Syntax
<i>filename</i>	Full directory path and filename that you want all coservers to use as a destination when they write conversion error messages	If you do not specify REJECTFILE, no reject files are created, and if errors occur, the load task will fail.	The filename must conform to the conventions of your operating system.
<i>field_delimiter</i>	Character to separate fields The default value is the pipe () character.	If you use a non-printing character as a delimiter, you must encode it as the octal representation of the ASCII character. For example, '\006' can represent CTRL-F.	Quoted String, p. 4-157

Element	Purpose	Restrictions	Syntax
<i>num_errors</i>	Number of errors allowed per coserver before the database server stops the load	If you do not set the MAXERRORS environment variable, the database server processes all data regardless of the number of errors. This parameter is ignored during an unload task.	Literal Number, p. 4-139
<i>num_rows</i>	Approximate number of rows contained in the external table	None.	Literal Number, p. 4-139
<i>record_delimiter</i>	Character to separate records If you do not set the RECORDEND environment variable, the default value is the newline character (\n).	If you use a non-printing character as a delimiter, you must encode it as the octal representation of the ASCII character. For example, '\006' can represent CTRL-F.	Quoted String, p. 4-157

(2 of 2)

Use the table options keywords as the following table describes. You can use each keyword whenever you plan to load or unload data unless only one of the two modes is specified.

Keyword	Purpose
CODESET	Specify the type of code set of the data
DEFAULT (load only)	Specify that the database server should replace missing values in delimited input files with column defaults (if they are defined) instead of inserting nulls This option allows input files to be sparsely populated. The input files do not need to have an entry for every column in the file where a default is the value to be loaded.
DELIMITER	Specify the character that separates fields in a delimited text file
DELUXE (load only)	Set a flag that causes the database server to load data in deluxe mode Deluxe mode is required for loading into STANDARD tables.

(1 of 2)

CREATE EXTERNAL TABLE

Keyword	Purpose
ESCAPE	Direct the database server to recognize ASCII special characters embedded in ASCII-text-based data files If you do not specify ESCAPE when you load data, the database server does not check the character fields in text data files for embedded special characters. If you do not specify ESCAPE when you unload data, the database server does not create embedded hexadecimal characters in text fields.
EXPRESS	Set a flag that causes the database server to attempt to load data in express mode If you request express mode but indexes or unique constraints exist on the table or the table contains BYTE or TEXT data, or the target table is not RAW or OPERATIONAL, the load stops with an error message reporting the problem.
FORMAT	Specify the format of the data in the data files
MAXERRORS	Set the number of errors that are allowed per coserver before the database server stops the load
RECORDEND	Specify the character that separates records in a delimited text file
REJECTFILE	Set the full pathname for all coservers to the area where reject files are written for data-conversion errors If conversion errors occur and you have not specified REJECTFILE or the reject files cannot be opened, the load job ends abnormally. For information on reject file naming and use of formatting characters, see “Reject Files” on page 2-103.
SIZE	Specify the approximate number of rows that are contained in the external table This option can improve performance if you use the external table in a join query.

(2 of 2)



Important: Check constraints on external tables are designed to be evaluated only when loading data. The database server cannot enforce check constraints on external tables because the data can be freely altered outside the control of the server. If you want to restrict rows that are written out to an external table during unload, use a *WHERE* clause to filter the rows.

Reject Files

Rows that have conversion errors during a load or rows that violate check constraints defined on the external table are written to a reject file on the coserver that performs the conversion. Each coserver manages its own reject file. The REJECTFILE keyword determines the name given to the reject file on each coserver.

You can use the formatting characters %c and %n (but not %r) in the filename format. If the reject files are written to the same disk, they must have unique filenames. Use the %c formatting characters to make the filenames unique. For more information on formatting characters, see the section [“Using Formatting Characters” on page 2-99](#).

If you perform another load to the same table during the same session, any earlier reject file of the same name is overwritten.

Reject file entries have the following format:

coserver-number, filename, record, reason-code, field-name: bad-line

Element	Purpose
<i>coserver-number</i>	Number of the coserver from which the file is read
<i>filename</i>	Name of the input file
<i>record</i>	Record number in the input file where the error was detected
<i>reason-code</i>	Description of the error
<i>field-name</i>	External field name where the first error in the line occurred, or '<none>' if the rejection is not specific to a particular column
<i>bad-line</i>	Line that caused the error (delimited or fixed-position character files only): up to 80 characters

The reject file writes the *coserver-number*, *filename*, *record*, *field-name* and *reason-code* in ASCII. The bad line information varies with the type of input file. For delimited files or fixed-position character files, up to 80 characters of the bad line are copied directly into the reject file. For Informix internal data files, the bad line is not placed in the reject file, because you cannot edit the binary representation in a file. However, *coserver-number*, *filename*, *record*, *reason-code*, and *field-name* are still reported in the reject file so you can isolate the problem.

The types of errors that cause a row to be rejected are as follows.

Error Text	Description
CONSTRAINT <i>constraint name</i>	This constraint was violated.
CONVERT_ERR	Any field encounters a conversion error.
MISSING_DELIMITER	No delimiter was found.
MISSING_RECORDEND	No recordend was found.
NOT NULL	A null was found in <i>field-name</i> .
ROW_TOO_LONG	The input record is longer than 32 kilobytes.

Examples

The examples in this section show how you can name files for use in the DATAFILES field.

Assume that the database server is running on four nodes, and one file is to be read from each node. All files have the same name. The DATAFILES item can then be as follows:

```
DATAFILES ("DISK:cogroup_all:/work2/unload.dir/mytbl")
```


Now, consider a system with 16 coservers where only three coservers have tape drives attached (for example, coservers 2, 5, and 9). If you define a cogroup for these coservers before you run load and unload commands, you can use the cogroup name rather than a list of individual coservers when you execute the commands. To set up the cogroup, run **onutil**.

```
% onutil
1> create cogroup tape_group
2> from coserver.2, coserver.5, coserver.9;
Cogroup successfully created.
```

Then define the file locations for named pipes.

```
DATAFILES ("PIPE:tape_group:/usr/local/TAPE.%c")
```

The filenames expand as follows:

```
DATAFILES ("pipe:2:/usr/local/TAPE.2",
           "pipe:5:/usr/local/TAPE.5",
           "pipe:9:/usr/local/TAPE.9")
```

If instead you want to process three files on each of two coservers, define the files as follows:

```
DATAFILES ("DISK:1:/work2/extern.dir/mytbl.%r(1..3)",
           "DISK:2:/work2/extern.dir/mytbl.%r(4..6)")
```

The expanded list is as follows:

```
DATAFILES ("disk:1:/work2/extern.dir/mytbl.1",
           "disk:1:/work2/extern.dir/mytbl.2",
           "disk:1:/work2/extern.dir/mytbl.3",
           "disk:2:/work2/extern.dir/mytbl.4",
           "disk:2:/work2/extern.dir/mytbl.5",
           "disk:2:/work2/extern.dir/mytbl.6")
```

References

Related Statements: INSERT, SELECT, and SET PLOAD FILE

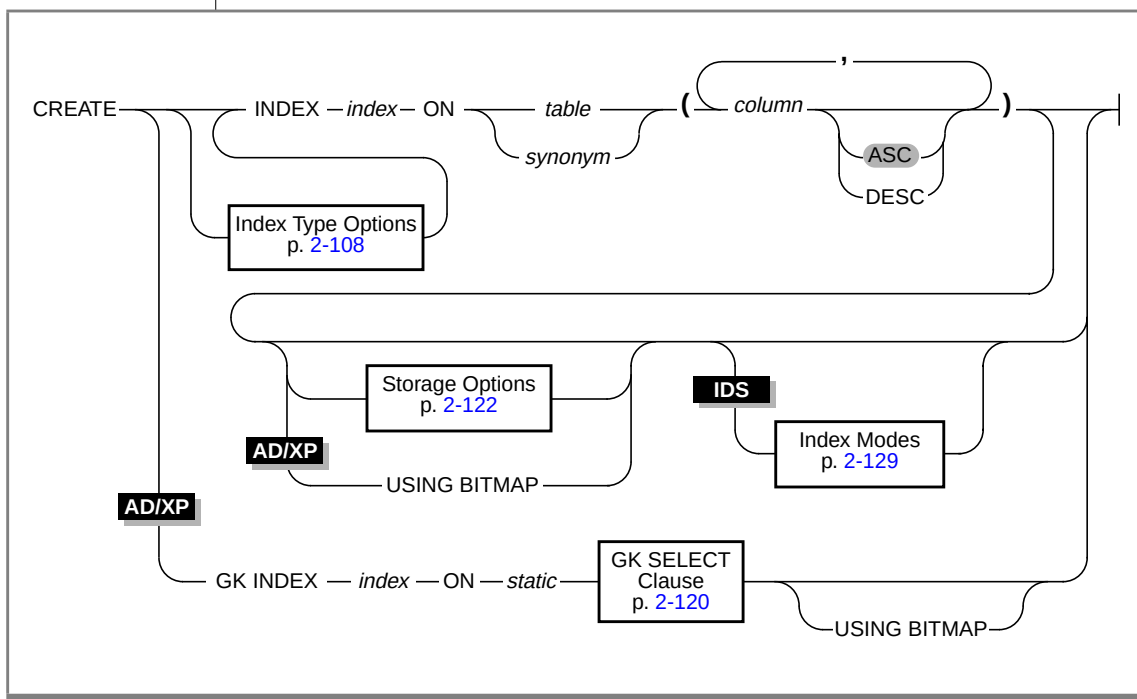
For more information on external tables, refer to your [Administrator's Guide](#).



CREATE INDEX

Use the CREATE INDEX statement to create an index for one or more columns in a table and, optionally, to cluster the physical table in the order of the index. When more than one column is 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.

Syntax



Element	Purpose	Restrictions	Syntax
<i>column</i>	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 2-110.	Identifier, p. 4-113
<i>index</i>	Name of the index to create	The name must be unique within the database. The first byte of the name cannot be a leading ASCII blank (hex 20).	Database Object Name, p. 4-25
<i>synonym</i>	Synonym for the name of the table on which the index is created	The synonym and the table to which the synonym points must already exist.	Database Object Name, p. 4-25
<i>static</i>	Name of the table on which a GK index is created	The table must exist. It must be a static table.	Database Object Name, p. 4-25
<i>table</i>	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.	Database Object Name, p. 4-25

Usage

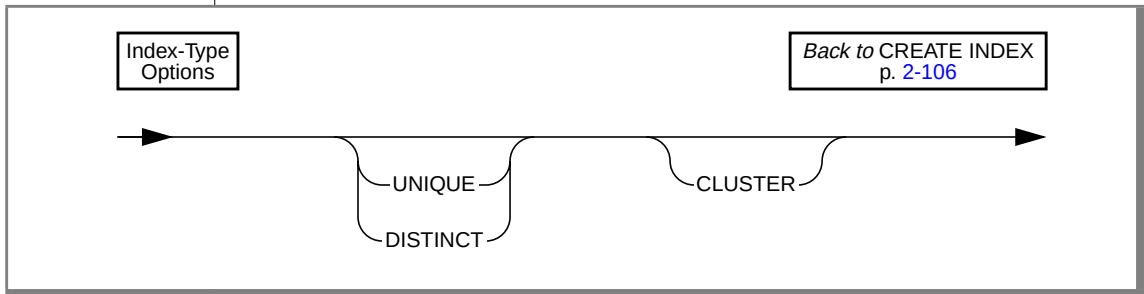
When you issue the CREATE INDEX statement, the table is locked in exclusive mode. If another process is using the table, the database server cannot execute the CREATE INDEX statement and returns an error.

AD/XP

If you are using Dynamic Server with AD and XP Options, use the USING BITMAP keywords to store the list of records in each key of the index as a compressed bitmap. The storage option is not compatible with a bitmap index because bitmap indexes must be fragmented in the same way as the table. ♦

Index-Type Options

The index-type options let you specify the characteristics of the index.



UNIQUE or DISTINCT Option

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 accomplishes 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.

AD/XP

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. For more information on creating unique constraints, see the CREATE TABLE or ALTER TABLE statements.

Restrictions for Attached Indexes in Dynamic Server with AD and XP Options

If a table is fragmented, any unique index must be defined on columns that define the fragmentation.

How Unique and Referential Constraints Affect Indexes

Internal indexes are created 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 that the index designates. 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.

AD/XP

If you are using Dynamic Server with AD and XP Options, you cannot use the CLUSTER option and storage options in the same CREATE INDEX statement (see [“Storage Options” on page 2-122](#)). When you create a clustered index the **constrid** of any unique or referential constraints on the associated table changes. The **constrid** is stored in the **sysconstraints** system catalog table. ♦

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 maximum number of columns and the total width of all columns vary with the database server. See [“Composite Indexes”](#).
- 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 2-111](#).
- You cannot add a unique index to a column or column list that has a primary-key constraint on it. The reason is that defining the column or column list as the primary key causes the database server to create a unique internal index on the column or column list. So you cannot create another unique index on this column or column list with the CREATE INDEX statement.
- The number of indexes you can create on the same column or same sequence of columns is restricted. See [“Number of Indexes Allowed” on page 2-116](#).

Composite Indexes

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.

You can include 16 columns in a composite index. The total width of all indexed columns in a single CREATE INDEX statement cannot exceed 255 bytes.

Place columns in the composite index in the order from most frequently used to least frequently used.

ASC and DESC Keywords

The ASC and DESC keywords allow you to specify whether the index is sorted in ascending or descending order. 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.

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 c_temp1 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 more information about leaf nodes in indexes, see your [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 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
:			
:			

(1 of 2)

stock_num	manu_code	description	unit_price
113	SHM	18-speed, assmbl'd	\$685.90
1	SMT	baseball gloves	\$450.00
6	SMT	tennis ball	\$36.00
5	SMT	tennis racquet	\$25.00

(2 of 2)

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);
```

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 2-111](#).

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 2-111](#).

Attached and Detached Indexes

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. *Attached indexes* are indexes created without a fragmentation strategy. Indexes are *detached indexes* when they are created with a fragmentation strategy or stored in separate dbspaces from the table.

Generalized-Key Indexes

If you are using Dynamic Server with AD and XP Options, you can create generalized-key (GK) indexes. Keys in a conventional index consist of one or more columns of the table being indexed. A GK index stores information about the records in a table based on the results of a query. Only tables created with the STATIC type can be used in a GK index.

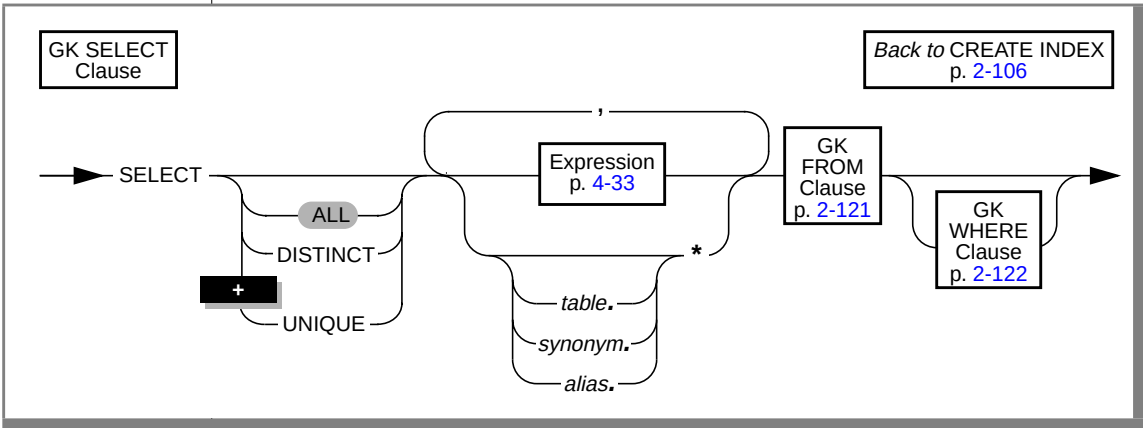
Generalized-key (GK) indexes provide a form of pre-computed index capability that allows faster query processing, especially in data-warehousing environments. The optimizer can use the GK index to increase performance.

A GK index is *defined on* a table when that table is the one being indexed. A GK index *depends on* a table when the table appears in the FROM clause of the index. Before you create a GK index, keep the following issues in mind:

- All tables used in a GK index must be static tables. If you try to change the type of a table to non-static while a GK index depends on that table, the database server returns an error.
- Since any table involved in a GK index needs to be a static type, UPDATE, DELETE, INSERT, and LOAD operations may not be performed on such a table until the dependent GK index is dropped and the table type changes.

SELECT Clause for Generalized-Key Index

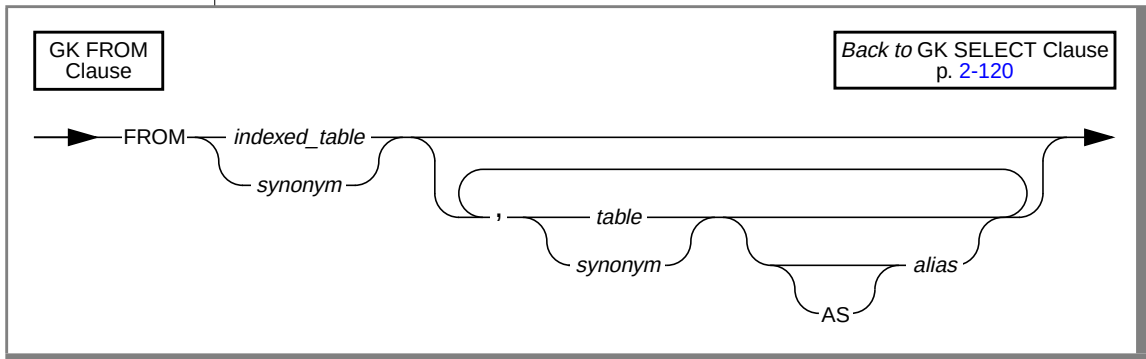
If you are using Dynamic Server with AD and XP Options, the options of the GK SELECT clause are a subset of the options of the SELECT statement on [page 2-450](#). The syntax of the GK SELECT clause has the same format as the syntax for the SELECT statement.



Element	Purpose	Restrictions	Syntax
<i>alias</i>	Temporary name assigned to the table in the FROM clause	You cannot use an alias for a SELECT clause unless you assign the alias to the table in the FROM clause. You cannot use an alias for the table on which the index is being built.	Identifier, p. 4-113
<i>synonym</i>	Name of the synonym from which you want to retrieve data	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table from which you want to retrieve data	The table must exist.	Database Object Name, p. 4-25

The following limitations apply to the expression in the GK SELECT clause:

- The expression cannot refer to any stored procedures.
- The expression cannot include the **USER**, **TODAY**, **CURRENT**, **DBINFO** functions or any function that refers to a time or a time interval.

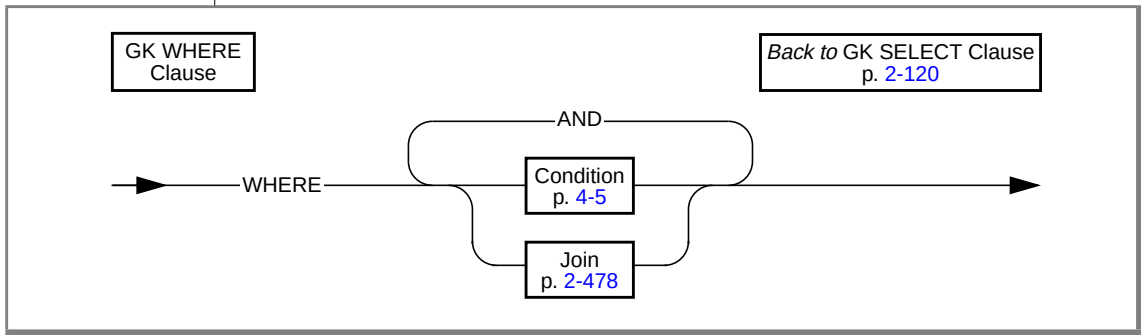
FROM Clause for Generalized-Key Index

Element	Purpose	Restrictions	Syntax
<i>alias</i>	Temporary name for a table	You cannot use an alias with <i>indexed_table</i> .	Identifier, p. 4-113
<i>indexed_table</i>	Name of the table on which the index is being built	The FROM clause must include the indexed table.	Database Object Name, p. 4-25
<i>synonym</i>	Synonym for the table from which you want to retrieve data	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table from which you want to retrieve data	The table must exist.	Database Object Name, p. 4-25

All tables that appear in the FROM clause must be local static tables. That is, no views, non-static, or remote tables are allowed.

The tables that are mentioned in the FROM clause must be *transitively joined on key* to the indexed table. Table **A** is transitively joined on key to table **B** if **A** and **B** are joined with equal-joins on the unique key columns of **A**. For example, suppose tables **A**, **B**, and **C** each have **col1** as a primary key. In the following example, **B** is joined on key to **A** and **C** is joined on key to **B**. **C** is transitively joined on key to **A**.

```
CREATE GK INDEX gki
(SELECT A.col1, A.col2 FROM A, B, C
 WHERE A.col1 = B.col1 AND B.col1 = C.col1)
```

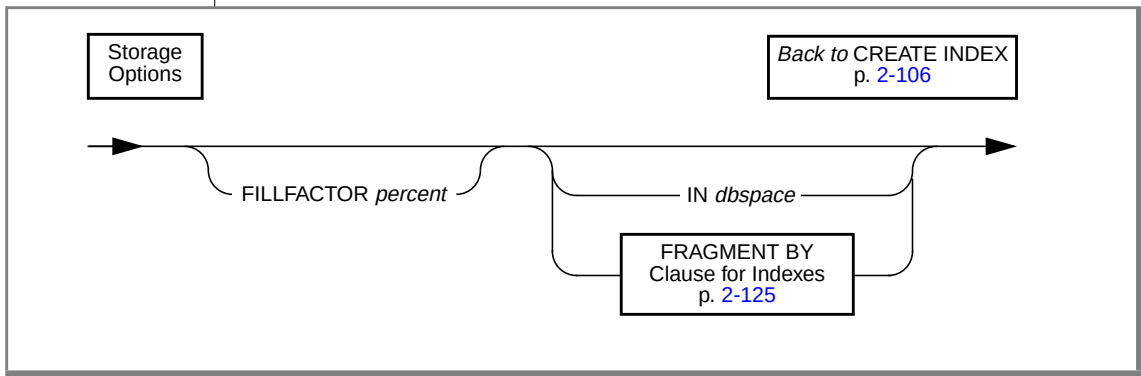
WHERE Clause for Generalized-Key Index

The WHERE clause for a GK index has the following limitations:

- The clause cannot include USER, TODAY, CURRENT, DBINFO functions or any functions that refer to time or a time interval.
- The clause cannot refer to any stored procedures.
- The clause cannot have any subqueries.
- The clause cannot use any aggregate function.
- The clause cannot have any IN, LIKE, or MATCH clauses.

Storage Options

The storage options let you specify how and where the index is stored.



Element	Purpose	Restrictions	Syntax
<i>dbspace</i>	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. 4-113
<i>percent</i>	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.	Literal Number, p. 4-139

AD/XP

If you are using Dynamic Server with AD and XP Options, you cannot create a clustered index that uses storage options. ♦

FILLFACTOR Clause

Use the FILLFACTOR clause to provide for expansion of the 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 index is created, the database 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 index. The nodes of the index initially fill to a certain percentage and contain space for inserts. The amount of available space depends on the number of keys in each page as well as the percentage value. For example, with a 50-percent FILLFACTOR value, the page would be half full and could accommodate doubling in 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.

This option takes effect only when you build an index on a table that contains more than 5,000 rows and uses more than 100 table pages, when you create an index on a fragmented table, or when you create a fragmented index on a nonfragmented table. 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 your [Administrator's Guide](#).

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.

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. For more information about fragmented indexes, see [“FRAGMENT BY Clause for Indexes” on page 2-125](#).

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 Clause for Indexes

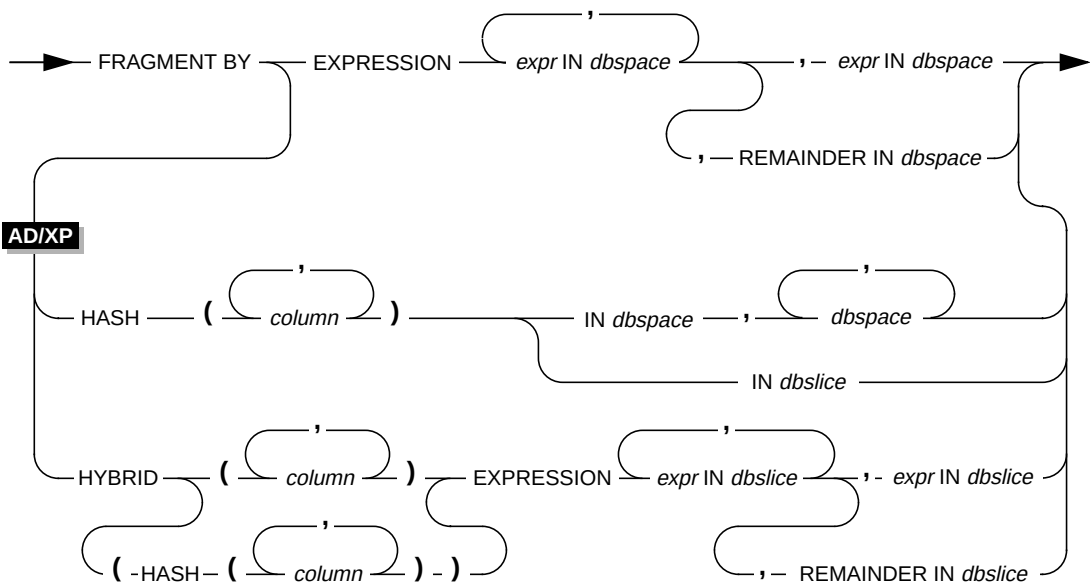
Use the FRAGMENT BY EXPRESSION clause to define the expression-based distribution scheme.

The FRAGMENT BY clause is not available with Dynamic Server, Workgroup and Developer Editions. ♦

W/D

FRAGMENT BY
Clause for Indexes

[Back to Storage Options](#)
p. 2-122



CREATE INDEX

Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of the column on which you want to fragment your index	All specified columns must be in the current table. If you specify a serial column, you cannot specify any other column.	Identifier, p. 4-113
<i>dbslice</i>	Name of the dbslice that contains all of the index fragments	The dbslice must exist when you execute the statement.	Identifier, p. 4-113
<i>dbspace</i>	Name of the dbspace that will contain an index fragment that <i>expr</i> defines	The dbspaces must exist at the time you execute the statement. You can specify a maximum of 2,048 dbspaces.	Identifier, p. 4-113
<i>expr</i>	Expression that defines a fragment where an index key is to be stored using a range, hash, or arbitrary rule	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 and / or time functions, or aggregates are allowed in the fragment expression.	Expression, p. 4-33 , and Condition, p. 4-5

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;

```

- 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. For more information on the **DBCENTURY** environment variable, see the [Informix Guide to SQL: Reference](#).

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 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 Unique Indexes with Dynamic Server

You can fragment unique indexes only with a table that uses an expression-based distribution scheme. The columns referenced in the fragment expression must be part of the indexed columns. If your CREATE INDEX statement fails to meet either of these restrictions, the CREATE INDEX fails, and work is rolled back.

Fragmentation of Indexes on Temporary Tables

You can 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.

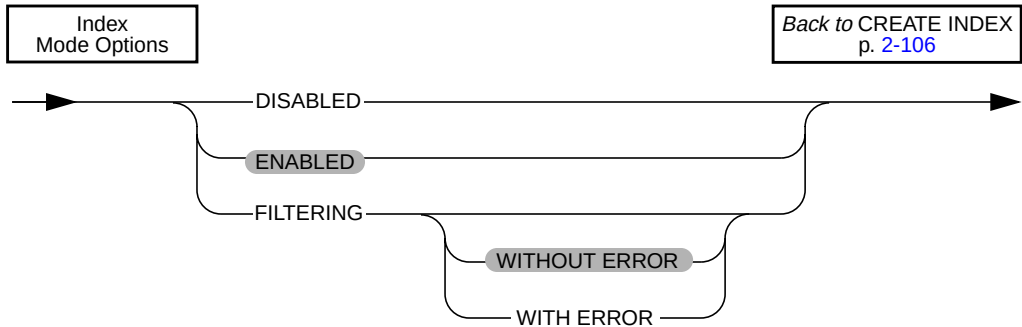
Instead of fragmenting the unique index on the temporary table, the database server creates the index in the first dbspace that the **DBSPACETEMP** environment variable specifies. To avoid this result, use the **FRAGMENT BY EXPRESSION** clause to specify a fragmentation strategy for the index.

For more information on the **DBSPACETEMP** environment variable, see the [Informix Guide to SQL: Reference](#).

IDS

Index Mode Options with Dynamic Server

Use the index-mode options to control the behavior of the index during insert, delete, and update operations.



The following table explains the index modes

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 query execution. 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 query execution. If an insert or update operation causes a duplicate key value to be added to a unique index in filtering mode, the statement continues processing, but the bad row is written to the violations table associated with the base table. Diagnostic information about the unique-index violation is written to the diagnostics table associated with the base table.

If you specify filtering, 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.
WITH ERROR	When a unique-index violation occurs during an insert or update operation, an integrity-violation error is returned to the user.

Specifying Modes for Unique Indexes

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

- You can set a unique index to enabled, disabled, or filtering.
- If you do not specify a mode, by default the index is enabled.
- If you do not specify the WITH ERROR or WITHOUT ERROR option, the default is WITHOUT ERROR.
- When you add a new unique index to an existing base table and specify the disabled mode for the index, your CREATE INDEX statement succeeds even if duplicate values in the indexed column would cause a unique-index violation.
- When you add a new unique index to an existing base table and specify the enabled or filtering mode for the index, your CREATE INDEX statement succeeds provided that no duplicate values exist in the indexed column that would cause a unique-index violation. However, if any duplicate values exist in the indexed column, your CREATE INDEX statement fails and returns an error.
- When you add a new unique index to an existing base table in the enabled or filtering mode, and duplicate values exist in the indexed column, erroneous rows in the base table are not filtered to the violations table. Thus, you cannot use a violations table to detect the erroneous rows in the base table.

Adding a Unique Index When Duplicate Values Exist in the Column

If you attempt to add a unique index in the enabled mode but receive an error message because duplicate values are in the indexed column, take the following steps to add the index successfully:

1. Add the index in the disabled mode. Issue the CREATE INDEX statement again, but this time specify the DISABLED keyword.
2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.

3. Issue a SET Database Object Mode statement to switch the mode of the index to enabled. When you issue this statement, existing rows in the target table that violate the unique-index requirement are duplicated in the violations table. However, you receive an integrity-violation error message, and the index remains disabled.
4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table. You might need to join the violations and diagnostics tables to get all the necessary information.
5. Take corrective action on the rows in the target table that violate the unique-index requirement.
6. After you fix all the nonconforming rows in the target table, issue the SET statement again to switch the disabled index to the enabled mode. This time the index is enabled, and no integrity violation error message is returned because all rows in the target table now satisfy the new unique-index requirement.

Specifying Modes for Duplicate Indexes

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

- You cannot set a duplicate index to the filtering mode.
- If you do not specify a mode of a duplicate index, by default the index is enabled.

IDS

How the Database Server Treats Disabled Indexes

In Dynamic Server, 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 on which the active constraint depends are disabled.

References

Related statements: ALTER INDEX, DROP INDEX, CREATE TABLE, and SET DATABASE OBJECT MODE.

For a discussion of the structure of indexes, see your [Administrator's Guide](#).

For information about performance issues with indexes, see your [Performance Guide](#).

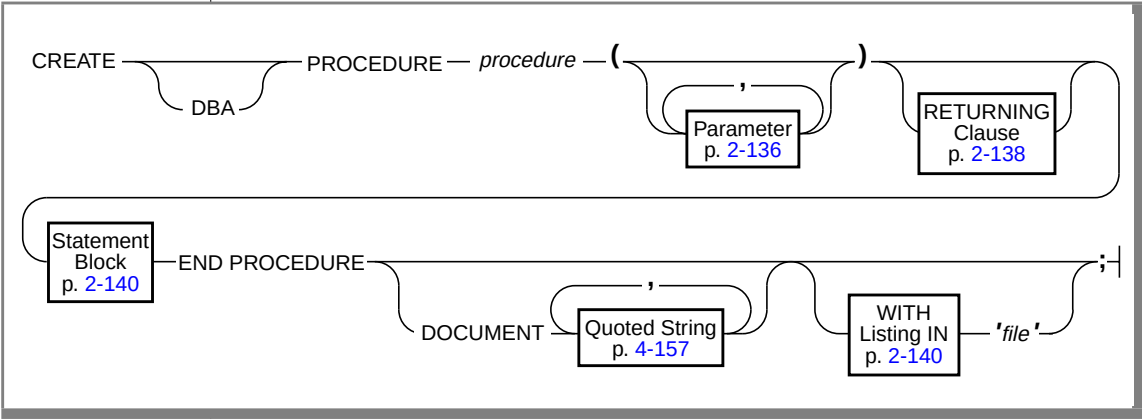
For a discussion of the GLS aspects of the CREATE INDEX statement, see the [Informix Guide to GLS Functionality](#). ♦

+

CREATE PROCEDURE

Use the CREATE PROCEDURE statement to name and define a stored procedure.

Syntax



Element	Purpose	Restrictions	Syntax
<i>file</i>	Listing file that is to contain warnings generated during the compilation of the procedure	The specified file must exist on the computer where the database resides.	The pathname and filename must conform to the conventions of your operating system.
<i>procedure</i>	Name of the procedure to create	The name must be unique in the database.	Database Object Name, p. 4-25

Usage

You must have the Resource privilege on a database to create a procedure.

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

E/C

If the statement block portion of the CREATE PROCEDURE statement is empty, no operation takes place when you call the procedure. You might use such a procedure in the development stage when you want to establish the existence of a procedure but have not yet coded it.

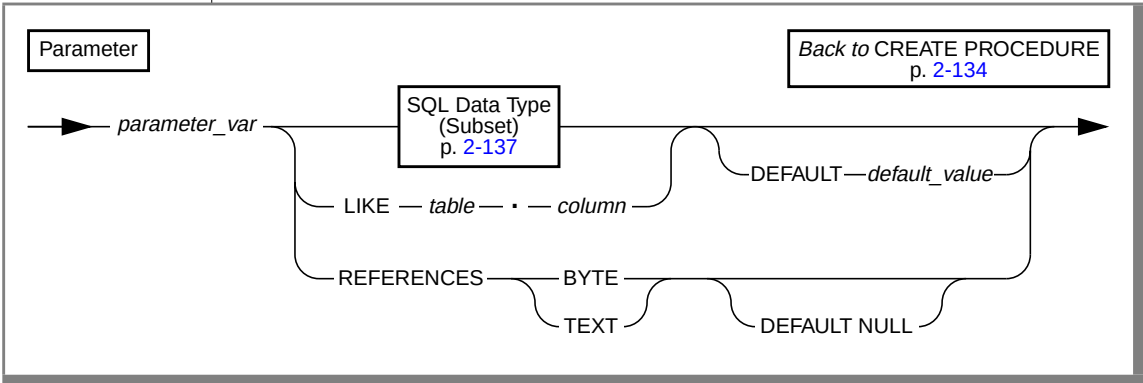
In ESQL/C, you can use a CREATE PROCEDURE statement only within a PREPARE statement. If you want to create a procedure for which the text is known at compile time, you must use a CREATE PROCEDURE FROM statement. ♦

DBA Option

If you create a procedure using the DBA option, it is known as a DBA-privileged procedure. If you do not use the DBA option, the procedure is known as an owner-privileged procedure. The privileges associated with the execution of a procedure are determined by whether the procedure is created with the DBA keyword. For more information, refer to the [Informix Guide to SQL: Tutorial](#).

If you create an owner-privileged procedure in a database that is not ANSI-compliant, the **NODEFDAC** environment variable prevents privileges on that procedure from being granted to PUBLIC. For further information on the **NODEFDAC** environment variable, see the [Informix Guide to SQL: Reference](#).

Parameter Syntax and Use



Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of a column whose data type is assigned to the <i>variable</i>	The column must exist in the specified table.	Identifier, p. 4-113
<i>default_value</i>	Default value that a procedure uses if you do not supply a value for <i>variable name</i> when you call the procedure	The default value must conform to the data type of <i>variable</i> .	Expression, p. 4-33
<i>parameter_var</i>	Name of a parameter for which you supply a value when you call the procedure	You can specify zero, one, or more parameters in a CREATE PROCEDURE statement.	Identifier, p. 4-113
<i>table</i>	Name of the table that contains <i>column</i>	The table must exist in the database.	Database Object Name, p. 4-25

To define a parameter within the CREATE PROCEDURE statement, you must specify its name. You must also specify its data type. You can specify the data type directly or use the LIKE or REFERENCES clauses to identify the data type.

Limits on Parameters

The number of parameters that you can define for a stored procedure is unlimited. However, the total length of all the parameters passed to a stored procedure must be less than 32 kilobytes.

Specifying a Default Value for a Parameter

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 procedure is called with fewer arguments than were defined for that procedure, the default value is used. If you do not provide a default value for a parameter, and the procedure is called with fewer arguments than were defined for that procedure, the calling application receives an error.

The following example shows a **CREATE PROCEDURE** statement that specifies a default value for a parameter. This procedure finds the square of the **i** parameter. If the procedure is called without specifying the argument for the **i** parameter, the database server uses the default value 0 for the **i** parameter.

```
CREATE PROCEDURE 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 PROCEDURE;
```

Subset of SQL Data Types Allowed in the Parameter List

The SQL Data Type subset includes all the SQL data types except **BYTE**, **TEXT**, and **SERIAL**. For the complete syntax of all the SQL data types, see page [4-27](#).

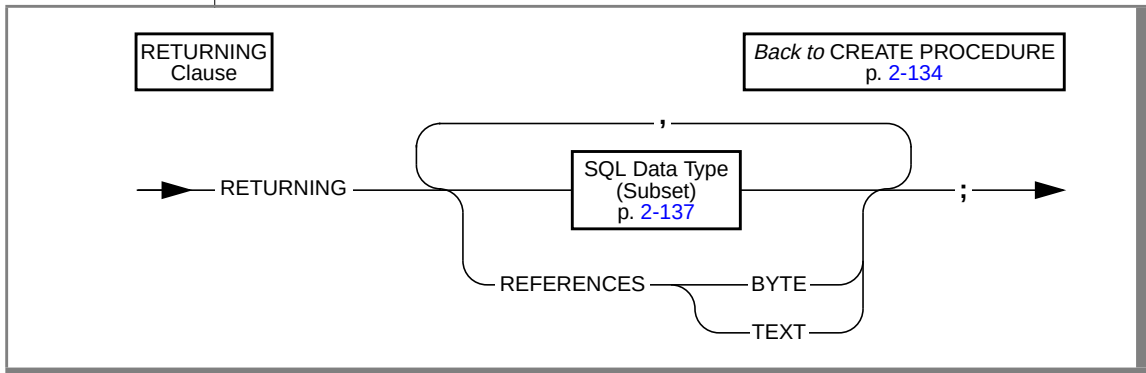
IDS

Referencing BYTE or TEXT Values

In Dynamic Server, use the **REFERENCES** clause to specify that a parameter contains **BYTE** or **TEXT** data. If you use the **DEFAULT NULL** option in the **REFERENCES** clause, and you call the procedure without a parameter, a null value is used.

RETURNING Clause

The RETURNING clause specifies the data type of each value you expect to be returned.



Once you use the RETURNING clause to indicate the type of values that are to be returned, you can use the RETURN statement at any point in the statement block to return procedure variables that correspond to the values in the RETURNING clause.

A procedure can return one or more sets of values. A procedure that returns more than one set of values (such as one or more columns from multiple rows of a table) is a cursory procedure. A procedure that returns only one set of values (such as one or more columns from a single row of a table) is a noncursory procedure.

The RETURNING clause can occur in a cursory procedure or in a noncursory procedure. In the following example, the RETURNING clause can return zero or one value if it occurs in a noncursory procedure. However, if this clause is associated with a cursory procedure, it returns more than one row from a table, and each returned row contains zero or one value.

```
RETURNING INT;
```

In the following example, the RETURNING clause can return zero or two values if it occurs in a noncursory procedure. However, if this clause is associated with a cursory procedure, it returns more than one row from a table and each returned row contains zero or two values.

```
RETURNING INT, INT;
```

In both of the preceding examples, the receiving procedure or program must be written appropriately to accept the information that is returned by the called procedure.

Limits on Return Values

The number of return values that you can define for a stored procedure is unlimited. However, the total length of all the values returned by the stored procedure must be less than 32 kilobytes.

Adding Comments to the Procedure

To add a comment to any line of a procedure, place a double-dash (--) before the comment or enclose the comment in curly brackets ({}). The double dash complies with the ANSI standard. The curly brackets are an Informix extension to the ANSI standard.

For examples of comments in procedures, see [“Specifying a Default Value for a Parameter” on page 2-137](#). For detailed information on the double-dash (--) and curly-brackets ({} symbols, see [“How to Enter SQL Comments” on page 1-6](#).

Describing the Procedure in the DOCUMENT Clause

The quoted string or strings in the DOCUMENT clause provide a synopsis and description of the procedure. The DOCUMENT text is intended for the user of the procedure. Anyone with access to the database can query the **sysprocbody** system catalog table to obtain a description of one or all the procedures stored in the database.

For example, to find the description of the procedure called **do_something**, execute the following query:

```
SELECT data FROM sysprocbody b, sysprocedures p
WHERE b.procid = p.procid      {join between the two tables}
AND
  p.procname = 'do_something'  {look for procedure do_something}
AND b.datakey = 'D'           { want user document }
```

WITH LISTING IN Option

The WITH LISTING IN option specifies a file where compile time warnings are sent. This listing file is created on the computer where the database resides. After you compile a procedure, this file holds one or more warning messages.

UNIX

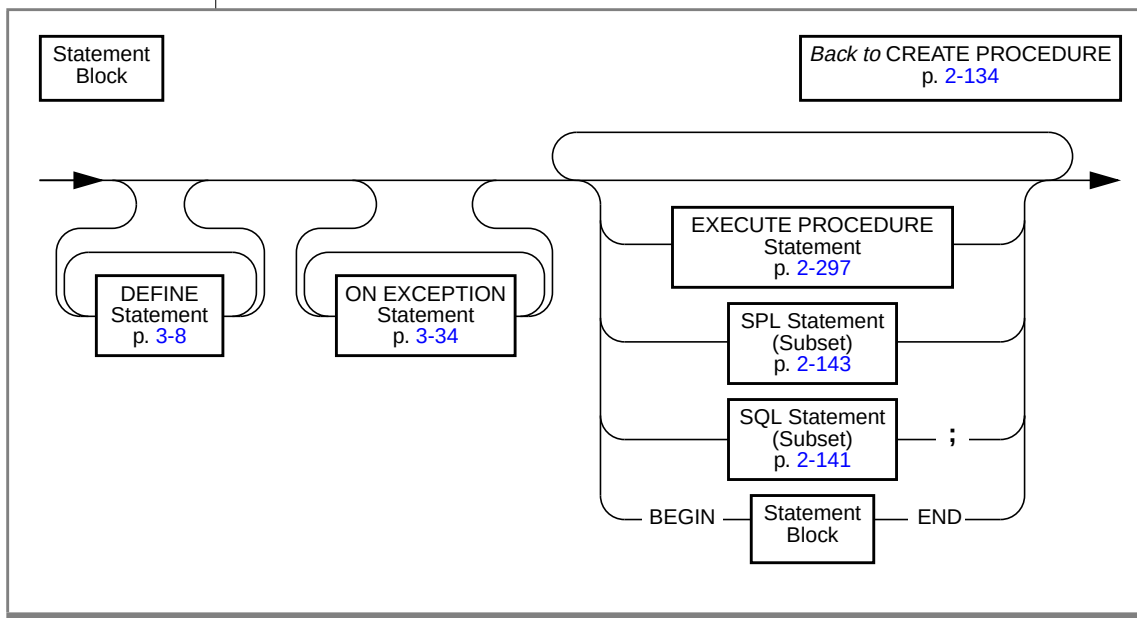
On UNIX, if you specify a filename but not a directory in the *file* variable, the default directory is your home directory on the computer where the database resides. If you do not have a home directory on this computer, the database server creates the file in the root directory. ♦

WIN NT

On Windows NT, if you specify a filename but not a directory in the *file* variable, the default directory is your current working directory if the database is on the local machine. Otherwise, the default directory is **%INFORMIXDIR%\bin.** ♦

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

Statement Block



The statement block can be empty, which results in a procedure that does nothing. Also, you cannot close the current database or select a new database within a procedure. And you cannot drop the current procedure within a procedure. You can, however, drop another procedure.

Subset of SQL Statements Allowed in the Statement Block

You can use any SQL statement in the statement block, except those in the following list:

CLOSE	FLUSH
CLOSE DATABASE	FREE
CONNECT	GET DESCRIPTOR
CREATE DATABASE	INFO
CREATE PROCEDURE	LOAD
CREATE PROCEDURE FROM	OPEN
DATABASE	OUTPUT
DECLARE	PREPARE
DESCRIBE	PUT
EXECUTE	SET DESCRIPTOR
EXECUTE IMMEDIATE	WHENEVER
FETCH	

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 procedure variables.

Support for Roles and User Identity

You can use roles with stored procedures. You can execute role-related statements (CREATE ROLE, DROP ROLE, and SET ROLE) and SET SESSION AUTHORIZATION statements within a stored procedure. You can also grant privileges to roles with the GRANT statement within a procedure. Privileges that a user has acquired through enabling a role or by a SET SESSION AUTHORIZATION statement are not relinquished when a procedure is executed.

CREATE PROCEDURE

For further information about roles, see the CREATE ROLE, DROP ROLE, GRANT, REVOKE, and SET ROLE statements in this guide.

Restrictions on a Procedure Called in a Data Manipulation Statement

If a stored procedure is called as part of an INSERT, UPDATE, DELETE, or SELECT statement, the called procedure cannot execute any statement in the following list. This restriction ensures that the stored procedure cannot make changes that affect the SQL statement that contains the procedure call.

ALTER FRAGMENT	DROP TABLE
ALTER INDEX	DROP TRIGGER
ALTER TABLE	DROP VIEW
BEGIN WORK	INSERT
COMMIT WORK	RENAME COLUMN
CREATE TRIGGER	RENAME TABLE
DELETE	ROLLBACK WORK
DROP DATABASE	UPDATE
DROP INDEX	
DROP SYNONYM	

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 the table. 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 the table.

Use of Transactions in Procedures

You can use the BEGIN WORK and COMMIT WORK statements in stored procedures. You can start a transaction, finish a transaction, or start and finish a transaction in a stored procedure. If you start a transaction in a stored procedure that is executed remotely, you must finish the transaction before the procedure exits.

Subset of SPL Statements Allowed in the Statement Block

You can use any of the following SPL statements in the statement block:

CALL	LET
CONTINUE	RAISE EXCEPTION
EXIT	RETURN
FOR	SYSTEM
FOREACH	TRACE
IF	WHILE

BEGIN-END Blocks

You can use the BEGIN keyword as the starting delimiter for a BEGIN-END block, and you can use the END keyword as the ending delimiter for a BEGIN-END block. As the syntax diagram in [“Statement Block” on page 2-140](#) shows, the BEGIN-END block can appear anywhere in the statement block that an SPL or SQL statement can appear.

You use the BEGIN-END block to define the scope of procedure variables and exception handlers. Variable declarations and exception handlers defined inside a BEGIN-END block are local to that block and are not accessible from outside the block.

References

Related statements: CREATE PROCEDURE FROM, DROP PROCEDURE, GRANT, EXECUTE PROCEDURE, PREPARE, UPDATE STATISTICS, and REVOKE

For a discussion of how to create and use stored procedures, see the [Informix Guide to SQL: Tutorial](#).

+

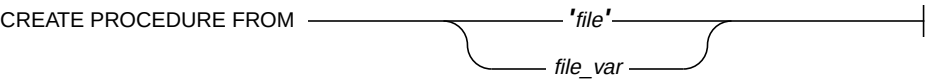
E/C

CREATE PROCEDURE FROM

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

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>file</i>	Name 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 file must contain only one CREATE PROCEDURE statement.	The pathname and filename must conform to the conventions of your operating system.
<i>file_var</i>	Name of a program variable that holds the value of <i>file</i>	The file that is specified in the program variable must exist. This file must contain only one CREATE PROCEDURE statement.	The name must conform to language-specific rules for variable names.

Usage

UNIX

On UNIX, if you specify a simple filename instead of a full pathname in the *file* parameter, the default directory is your home directory on the computer where the database resides. If you do not have a home directory on this computer, the default directory is the root directory. ♦

WIN NT

On Windows NT, if you specify a filename but not a directory in the *file* parameter, the default directory is your current working directory if the database is on the local computer. Otherwise, the default directory is %INFORMIXDIR%\bin. ♦

The file that you specify in the *file* parameter can contain only one CREATE PROCEDURE statement.

References

Related statements: CREATE PROCEDURE statement

For a discussion of how to create and use stored procedures, see the [Informix Guide to SQL: Tutorial](#).

+

IDS

CREATE ROLE

Use the CREATE ROLE statement to create a new role.

You can use this statement only with Dynamic Server.

Syntax

```
CREATE ROLE _____ role _____
```

Element	Purpose	Restrictions	Syntax
role	Name assigned to a role created by the DBA	Maximum number of characters is eight. 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. 4-113

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

Related statements: DROP ROLE, GRANT, REVOKE, and SET ROLE

For a discussion of how to use roles, see the [Informix Guide to Database Design and Implementation](#).

DB

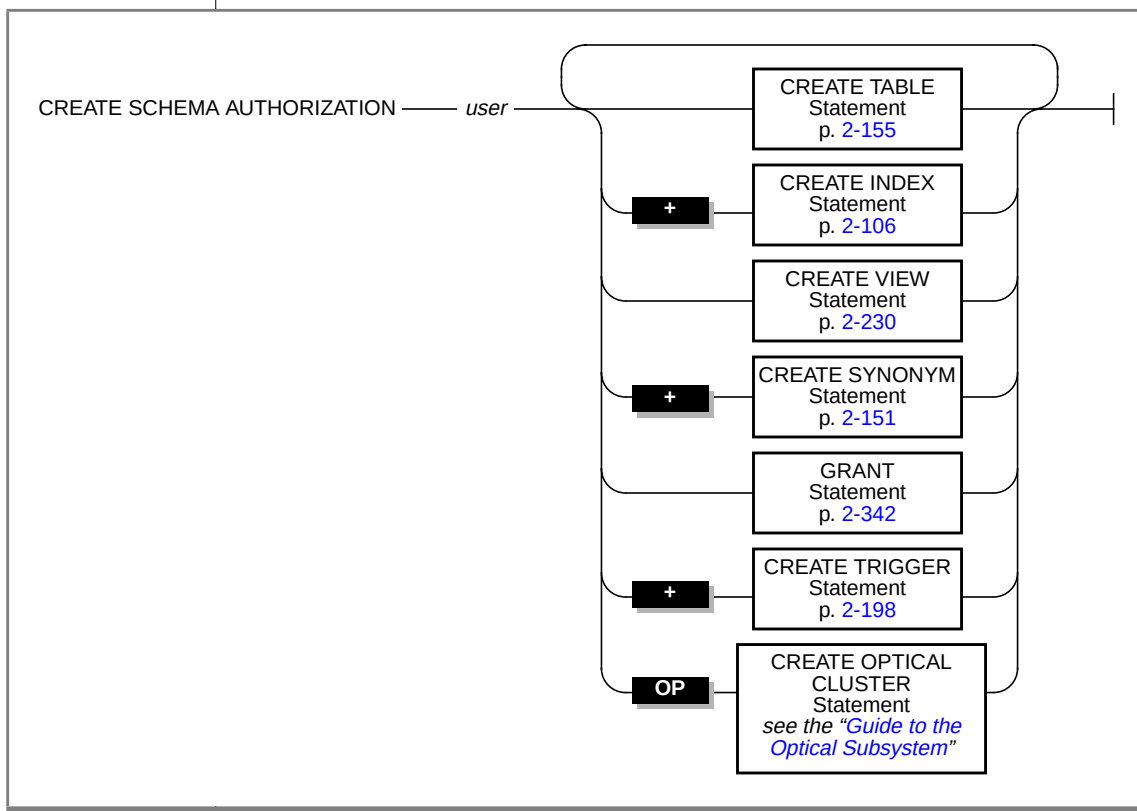
SQLE

CREATE SCHEMA

Use the CREATE SCHEMA statement to issue a block of CREATE and GRANT statements as a unit. The CREATE SCHEMA statement allows you to specify an owner of your choice for all database objects that the CREATE SCHEMA statement creates.

Use this statement with DB-Access and SQL Editor.

Syntax



Element	Purpose	Restrictions	Syntax
<i>user</i>	Name of the user who owns the database objects that the CREATE SCHEMA statement creates	If the user who issues the CREATE SCHEMA statement has the Resource privilege, <i>user</i> must be the name of this user. If the user who issues the CREATE SCHEMA statement has the DBA privilege, <i>user</i> can be the name of this user or another user.	Identifier, p. 4-113

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* must be the name of the person with the Resource privilege who is running the CREATE SCHEMA statement. Anyone with the DBA privilege can also create a schema for someone else. In this case, *user* can identify a user other than the person who is running the CREATE SCHEMA statement.

You can put CREATE and GRANT statements in any logical order 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 > '12/31/1997'
  CREATE INDEX idxtime ON mytable (mytime);
```

Creating Database Objects Within CREATE SCHEMA

All database objects that a CREATE SCHEMA statement creates are owned by *user*, even if you do not explicitly name each database object. If you are the DBA, you can create database objects for another user. If you are not the DBA, and you try to create a database 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 Database Objects or Granting Privileges Outside CREATE SCHEMA

If you create a database object or use the GRANT statement outside a CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set **DBANSIWARN**.

References

Related statements: CREATE INDEX, CREATE SYNONYM, CREATE TABLE, CREATE VIEW, and GRANT

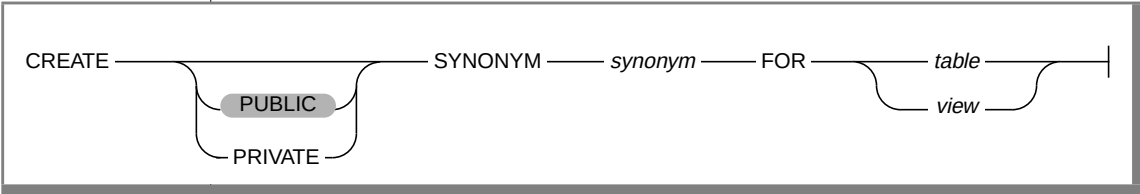
For a discussion of how to create a database, see the [Informix Guide to Database Design and Implementation](#).

+

CREATE SYNONYM

Use the CREATE SYNONYM statement to provide an alternative name for a table or view.

Syntax



Element	Purpose	Restrictions	Syntax
<i>synonym</i>	Name of the synonym to be created	The synonym name must be unique in the database.	Database Object Name, p. 4-25
<i>table</i>	Name of the table for which the synonym is created	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view for which the synonym is created	The view must exist.	Database Object Name, p. 4-25

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

In an ANSI-compliant database, 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, the database server verifies that the database object referred to by the synonym exists when you create the synonym.

The following example shows how to create a synonym for a database 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 in an Informix database server earlier than Version 5.0 is a public synonym.

ANSI

In an ANSI-compliant database, 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 **acctng**, 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 **acctng** 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 **acctng**, 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 **acctng** database server now points to a new version of the **customer** table on the **training** database server.

References

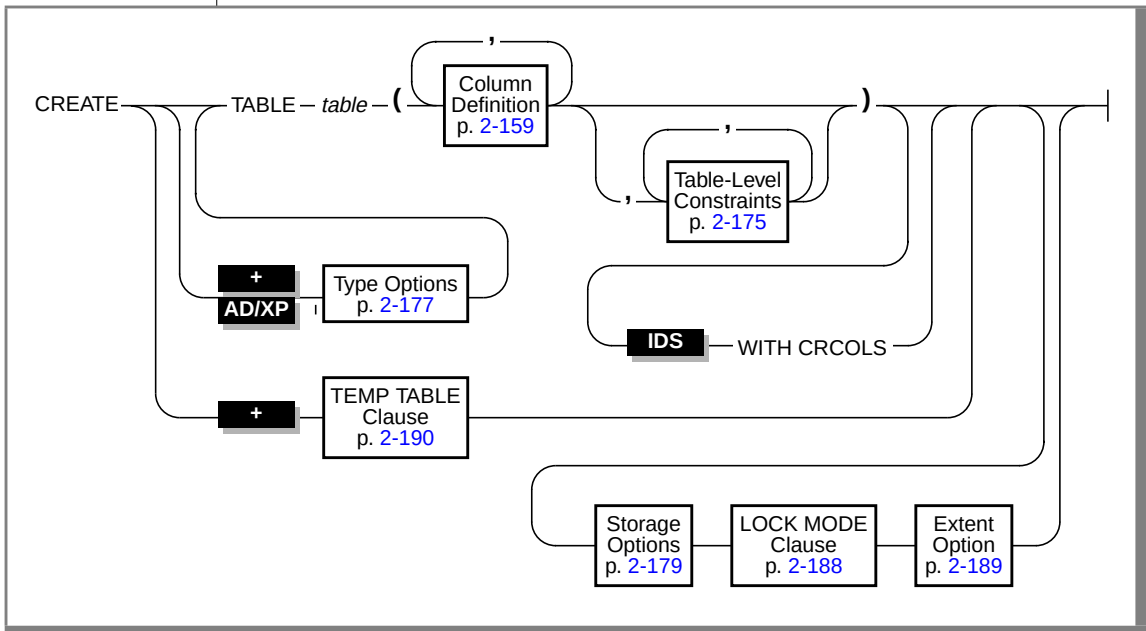
Related statement: DROP SYNONYM

For a discussion of concepts related to synonyms, see the [Informix Guide to Database Design and Implementation](#).

CREATE TABLE

Use the CREATE TABLE statement to create a new table in the current database, place data-integrity constraints on the table, designate the size of its initial and subsequent extents, fragment the table, and specify the locking mode. You can create permanent or temporary tables.

Syntax



Element	Purpose	Restrictions	Syntax
<i>table</i>	Name that you want to assign to the table	The name must be different from any existing table, view, or synonym name in the current database.	Database Object Name, p. 4-25

DB

E/C

ANSI

Usage

Table names must be unique within a database. 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 other temporary table names used by other users.

In DB-Access, using the CREATE TABLE statement outside the CREATE SCHEMA statement generates warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

In ESQL/C, using the CREATE TABLE statement generates warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

Privileges on Tables

By default, all users who have been granted the Connect privilege to a database have all access privileges (except Alter and References) to the new table. To further restrict access, use the REVOKE statement to take *all* access away from PUBLIC (everyone on the system). Then, use the GRANT statement to designate the access privileges that you want to give to specific users.

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 how to prevent privileges from being granted to PUBLIC, see the **NODEFDAC** environment variable in the [Informix Guide to SQL: Reference](#).

In an ANSI-compliant database, no default table-level privileges exist. You must grant these privileges explicitly. ♦

Defining Constraints

When you create a table, several elements must be defined. For example, the table and columns within that table must have unique names. Also, every table column must have at least a data type associated with it. You can also, optionally, place several constraints on a given column. For example, 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.



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 Database Design and Implementation](#). For information on how to check effectively, see the SET Database Object Mode statement on [page 2-512](#).

Important: *In a database without logging, detached checking is the only constraint-checking mode available. Detached checking means that constraint checking is performed on a row-by-row basis.*

You can define constraints at either the *column* or *table* level. If you choose to define constraints at the column level, you cannot have multiple-column constraints. In other words, the constraint created at the column level can apply only to a single column. If you choose to define constraints at the table level, you can apply constraints to single or multiple columns. At either level, single-column constraints are treated the same way.

Whenever you place a data restriction on a column, a constraint is created automatically. You have the option of specifying a name for the constraint. If you choose not to specify a name for the constraint, the database server creates a default constraint name for you automatically.

When you create a constraint of any type, the name of the constraint must be unique within the database.

ANSI

In an ANSI-compliant database, when you create a constraint of any type, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within the database. ♦

Limits on Constraint Definitions

You can include 16 columns in a list of columns for a unique, primary-key, or referential constraint. The total length of all columns cannot exceed 255 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 2-594](#).

Adding or Dropping Constraints

After you have used the CREATE TABLE statement to place constraints on a column or set of columns, you must use the ALTER TABLE statement to add or drop the constraint from the column or composite column list.

Enforcing Primary-Key, Unique, and Referential Constraints

Primary-key, unique, and referential constraints are implemented as an ascending index that allows only unique entries or an ascending index that allows duplicates. When one of these constraints is placed on a column, the database server performs the following functions:

- Creates a unique index for a unique or primary-key constraint
- Creates a non-unique 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 constraints. Instead, the existing index is *shared* by the constraints. If the existing index is non-unique, it is *upgraded* 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 data 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

A row is added to the **sysindexes** system catalog table for each new primary-key, unique, or referential constraint that does not share an index with an existing constraint. The index name in the **sysindexes** system catalog table is created with the following format:

```
[space]<tabid>_<constraint id>
```

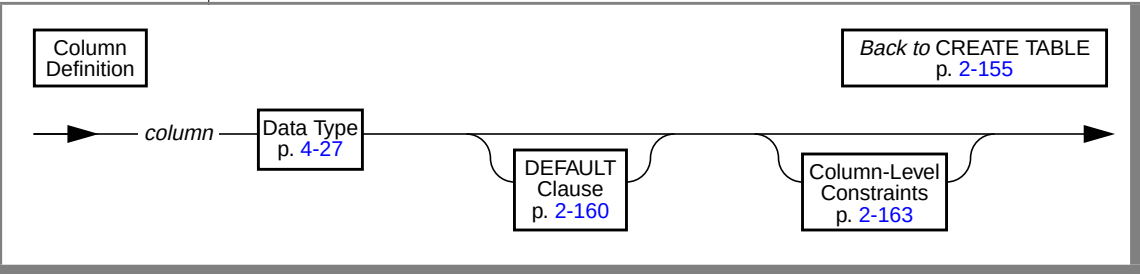
In this format, *tabid* and *constraint id* are from the **systables** and **sysconstraints** system catalog tables, respectively. For example, the index name might be something like this: " **121_13**" (quotes used to show the space).

The constraint name must be unique within the database. If you do not specify a *constraint name*, the database server generates one for the **sysconstraints** system catalog table using the following template:

```
<constraint type><tabid>_<constraint id>
```

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.

Column Definition

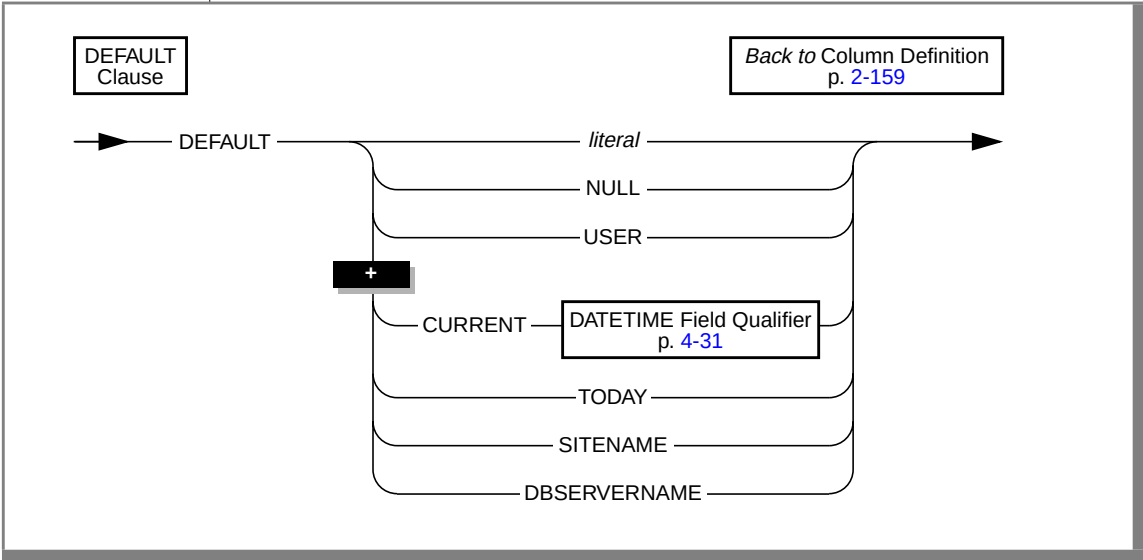


Element	Purpose	Restrictions	Syntax
column	Name of a column in the table	Name must be unique within a table, but you can use the same names in different tables in the same database.	Identifier, p. 4-113

Use the column-definition portion of the CREATE TABLE statement to list the name, data type, default values, and constraints *of a single column* as well as to indicate whether the column does not allow duplicate values.

DEFAULT Clause

The default value is inserted into the column when an explicit value is not specified.



Element	Purpose	Restrictions	Syntax
<i>literal</i>	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 as Default Values ” on page 2-161.	Expression, p. 4-33

NULL as the Default Value

If a default is not specified, and the column allows nulls, the default is NULL. If you designate NULL as the default value for a column, you cannot specify a not null constraint as part of the column definition.

You cannot designate default values for serial columns. If the column is BYTE or TEXT data type, you can *only* designate NULL as the default value.

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
DECIMAL	DECIMAL, MONEY, FLOAT, SMALLFLOAT
CHARACTER	CHAR, NCHAR, NVARCHAR, VARCHAR, DATE
INTERVAL	INTERVAL
DATETIME	DATETIME

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.

For information on how to use a literal INTERVAL, refer to the Literal INTERVAL segment on [page 4-136](#). For more information on how to use a literal DATETIME, refer to the Literal DATETIME segment on [page 4-133](#).

You cannot designate NULL as a default value for a column that is part of a primary key.

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, NCHAR, NVARCHAR, or VARCHAR column at least 18 characters long
SITENAME	CHAR, NCHAR, NVARCHAR, or VARCHAR column at least 18 characters long
TODAY	DATE column
USER	CHAR or VARCHAR column at least 8 characters long

For more information about the options that you can use, refer to [“Constant Expressions” on page 4-47](#).

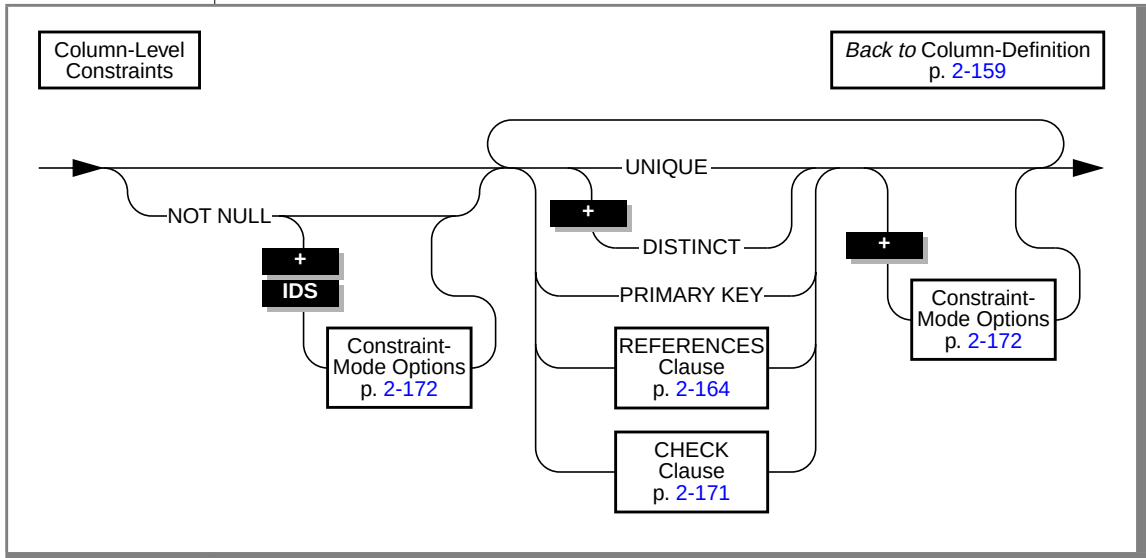
Example of Default Values in Column Definitions

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)
```

Column-Level Constraints

Constraints at the column level are limited to a single column. In other words, you cannot use constraints that involve more than one column. For information on multiple-column constraints, see [“Table-Level Constraints” on page 2-175](#).



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))
```

Using the Not-Null Constraint

If you do not indicate a default value for a column, the default is null *unless* you place a not-null constraint on the column. In that case, no default value exists for the column. 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))
```

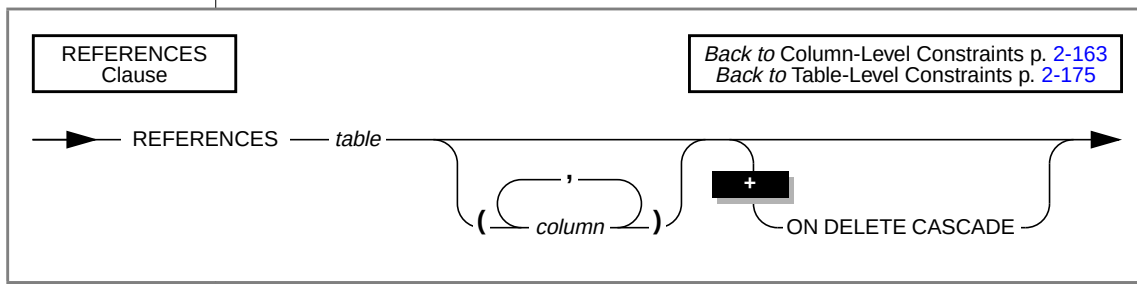
If you place a not-null constraint on a column (and no default value is specified), you *must* enter a value into this column when you insert a row or update that column in a row. If you do not enter a value, the database server returns an error.

Using Byte and Text Data 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 not-null values if you place a check constraint on a BYTE or TEXT column.

REFERENCES Clause

The REFERENCES clause allows you to place a foreign-key reference on a column. You can establish a referential relationship between two tables or within the same table.



Element	Purpose	Restrictions	Syntax
<i>column</i>	Referenced column or columns in the referenced table	See “ Restrictions on the Column Variable in the REFERENCES Clause ” on page 2-165.	Identifier, p. 4-113
<i>table</i>	Name of the referenced table	The referenced table must reside in the same database as the referencing table.	Database Object Name, p. 4-25

The referenced column (the column that you specify in the *column name* variable) can be in the same table as the referencing column, or the referenced column can be in a different table in the same database.

If the referenced table is different from the referencing table, the default column (or columns) is the primary-key column (or columns) of the referenced table. If the referenced table is the same as the referencing table, there is no default.

Restrictions on the Column Variable in the REFERENCES Clause

You must observe the following restrictions on the *column* variable in the REFERENCES clause:

- The referenced column 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 on it.
- The data type of the referenced column must be identical to the data type of the referencing column. The only exception is that a referencing column must be INTEGER if the referenced column is SERIAL.
- You can specify only one column when you are using the REFERENCES clause at the column level (that is, when you are using the REFERENCES clause in the Column-Definition option).

You can specify multiple columns when you are using the REFERENCES clause at the table level (that is, when you are using the REFERENCES clause in the Constraint-Definition option). You can specify a maximum of 16 column names. The total length of all the columns cannot exceed 255 bytes.

Referenced and Referencing Column Requirements

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. In other words, the referenced column or set of columns must be a unique or primary-key constraint. If the referenced columns do not meet this criteria, the database server returns an error.

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). This parent-child relationship is established through a referential constraint.

A referential constraint can be established between two tables or within the same table. 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).

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. The references section of the previous example can be rewritten, as the following example shows:

```

      .
      .
      .
      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.

Data Type Restrictions

The data types of the referencing and referenced columns must be identical unless the column is SERIAL data type. You can specify SERIAL for the primary key of the parent table and INTEGER for the foreign key. In the previous example, a one-to-one correspondence exists between the data types of the primary and foreign keys. If the primary-key column was defined as type SERIAL, the statement would still be successfully executed.

You cannot place a referential constraint on a BYTE or TEXT column.

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 REFERENCES in a Column Definition

When you use the REFERENCES clause at the column-definition level, you can reference a single column. 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))
```

Note that **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 you are referencing the primary key in another table, you do not need to specify the referenced table 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 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 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.

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. After you indicate cascading deletes, when you delete a row from a parent table, the database server deletes any associated matching rows from the child table.

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.

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.

Restrictions on Cascading Deletes

Cascading deletes can be used for most deletes. One 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.

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

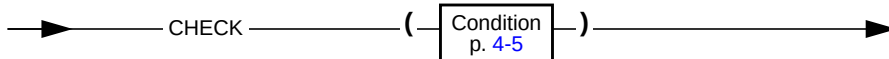
For a detailed discussion of cascading deletes, see the [Informix Guide to SQL: Tutorial](#).

CHECK Clause

The check clause allows 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
Clause

[Back to Column-Level Constraints p. 2-163](#)
[Back to Table-Level Constraints p. 2-175](#)



You use *search conditions* to define check constraints. The search condition cannot contain subqueries; aggregates; host variables; rowids; the CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY functions; or stored procedure calls.

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 **acct_chk** has two columns with check constraints:

```
CREATE TABLE acct_chk (
  chk_id SERIAL PRIMARY KEY,
  debit INTEGER REFERENCES accounts (acc_num),
  debit_amt MONEY CHECK (debit_amt BETWEEN 0 AND 99999),
  credit INTEGER REFERENCES accounts (acc_num),
  credit_amt MONEY CHECK (credit_amt BETWEEN 0 AND 99999))
```

Both **debit_amt** and **credit_amt** are columns of MONEY data type whose values must be between 0 and 99999. If, however, you wanted to test that both columns contained the same value, 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 a check constraint is defined 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 acct_chk (
  chk_id SERIAL PRIMARY KEY,
  debit INTEGER REFERENCES accounts (acc_num),
  debit_amt MONEY,
  credit INTEGER REFERENCES accounts (acc_num),
  credit_amt MONEY,
  CHECK (debit_amt = credit_amt))
```

In this example, the **debit_amt** and **credit_amt** columns must equal each other, or the insert or update fails.

Constraint-Mode Options

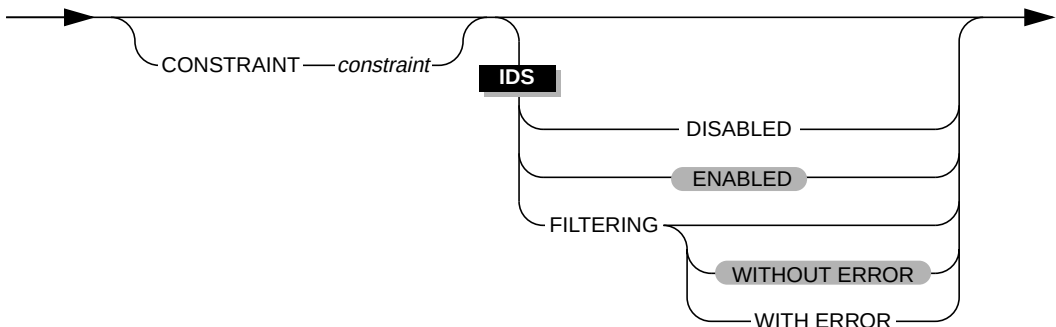
Use the constraint mode options for the following purposes:

- To assign a name to a constraint.
- To set a constraint to the disabled, enabled, or filtering mode in Dynamic Server. ♦

IDS

Constraint-Mode
Options

[Back to Column-Level Constraints p. 2-163](#)
[Back to Table-Level Constraints p. 2-175](#)



Element	Purpose	Restrictions	Syntax
<i>constraint</i>	Name of the constraint	The constraint name must be unique within the database.	Database Object Name, p. 4-25

Using the CONSTRAINT Clause

The CONSTRAINT clause allows you to assign a meaningful name to a constraint. If you do not name a constraint, the database server assigns an identifier similar to u143_2. Constraint names appear in error messages having to do with constraint violations. In addition, constraint names are specified in the DROP CONSTRAINT clause of the ALTER TABLE statement.

IDS

Description of Constraint-Mode Options

In Dynamic Server, use the constraint-mode options to control the behavior of constraints during insert, delete, and update operations. The following list explains these options.

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.

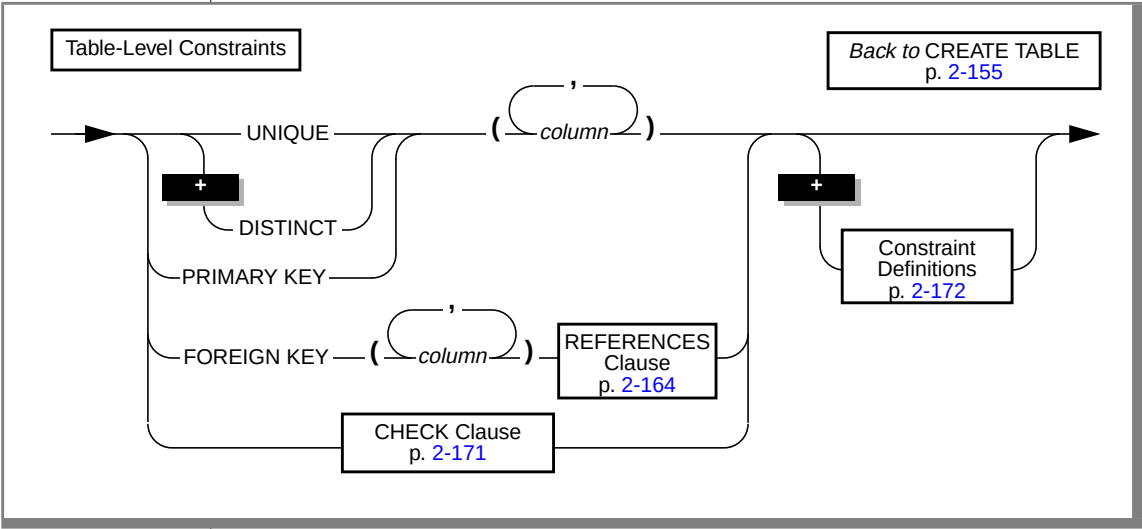
Specifying Modes for Constraints

You must observe the following rules when you specify modes for constraints:

- If you do not specify a mode, by default the column-level constraint or table-level constraint is enabled.
- If you do not specify the `WITH ERROR` or `WITHOUT ERROR` option for a filtering-mode constraint, the default is `WITHOUT ERROR`.
- Constraints defined on temporary tables are always enabled. You cannot create a constraint on a temporary table in the disabled or filtering mode, nor can you use the `SET Database Object Mode` statement to switch the mode of a constraint 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.

Table-Level Constraints

The table-level constraints definitions allow you to create constraints for a single column or a set of columns.



Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of the column or columns on which the constraint is placed	The column cannot be a BYTE or TEXT column.	Identifier, p. 4-113

The *column* must be a column in the table. You can include up to 16 columns in a list of columns. The total length of the list of columns cannot exceed 255 bytes.

Defining a Column as Unique

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.

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 (a_name CHAR(12), a_code SERIAL,  
    UNIQUE (a_name) CONSTRAINT acc_name)
```

If you want to define the constraint at the column level instead, simply include the keywords `UNIQUE` and `CONSTRAINT` in the column definition, as the following example shows:

```
CREATE TABLE accounts  
    (a_name CHAR(12) UNIQUE CONSTRAINT acc_name, a_code SERIAL)
```

Restrictions for Unique Constraints

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.

Defining a Column as a Primary Key

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 **a_name**. The following example creates this column as the primary key for the **accounts** table:

```
CREATE TABLE accounts  
    (a_name CHAR(12), a_code SERIAL, PRIMARY KEY (a_name))
```

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.

Defining a Column as a Foreign Key

A foreign key, or referential constraint, 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. Restrictions for Referential Constraints

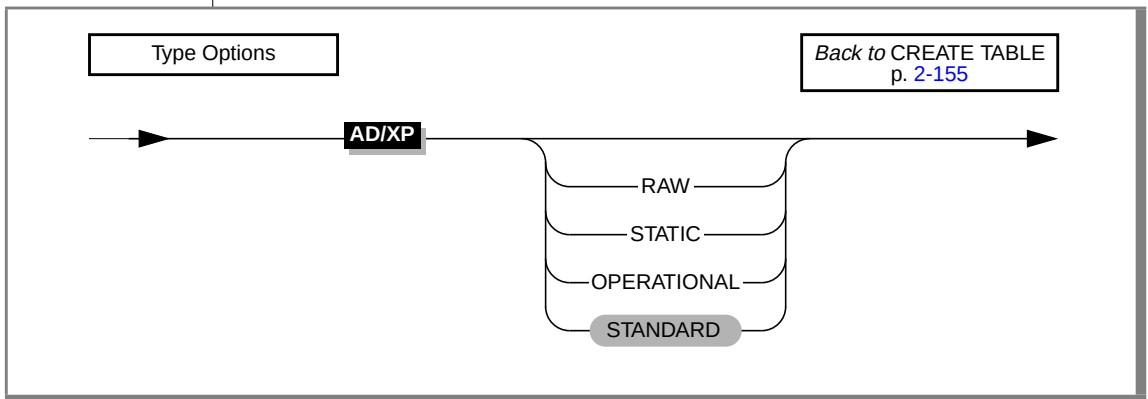
Restrictions for Referential Constraints

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.

AD/XP

Type Options

In Dynamic Server with AD and XP Options, the type options let you specify the logging characteristics of a table.



When you have Dynamic Server with AD and XP Options, a permanent table can be any of the four following types:

- RAW, a non-logging table that uses light appends
- STATIC, a non-logging table that can contain index and referential constraints

- OPERATIONAL, a logging table that uses light appends and is not restorable from archives
- STANDARD (default type of permanent table), a logging table that allows rollback, recovery, and restoration from archives

For a more detailed description of these table types, refer to the [Informix Guide to Database Design and Implementation](#).

WITH CRCOLS Keywords

In Dynamic Server, the WITH CRCOLS keywords create two shadow columns that Enterprise Replication uses for conflict resolution. The first column, **cdrserver**, contains the identity of the database server where the last modification occurred. The second column, **cdrtime**, contains the time stamp of the last modification. You must add these columns before you can use time-stamp or stored-procedure conflict resolution.

For most database operations, the **cdrserver** and **cdrtime** columns are hidden. For example, if you include the WITH CRCOLS keywords when you create a table, the **cdrserver** and **cdrtime** columns:

- do not appear when you issue the statement
`SELECT * from tablename`
- do not appear in DB-Access when you ask for information about the columns of the table.
- are not included in the number of columns (**ncols**) in the **systables** system catalog table entry for *tablename*.

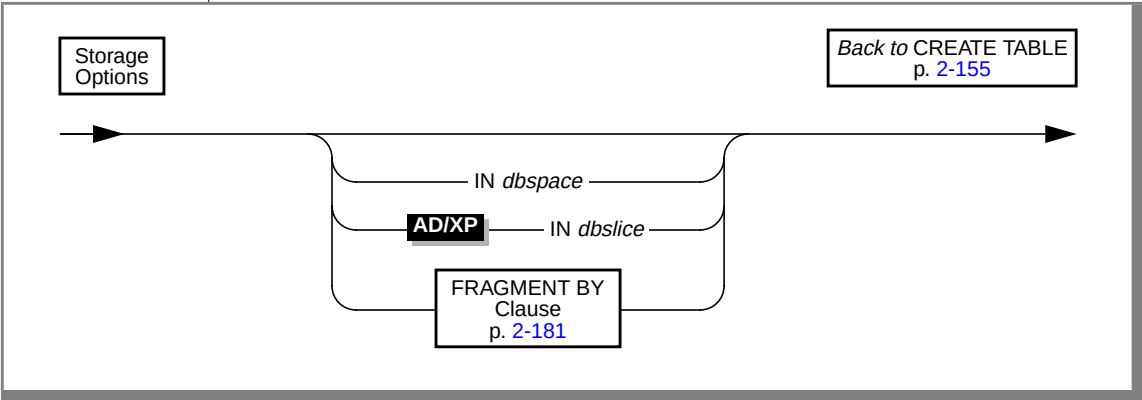
To view the contents of **cdrserver** and **cdrtime**, explicitly name the columns in a SELECT statement:

```
SELECT cdrserver, cdrtime from tablename
```

For more information about using WITH CRCOLS, refer to the [Guide to Informix Enterprise Replication](#).

Storage Options

The storage options allow you to specify where the table is stored and the fragmentation strategy for the table.



Element	Purpose	Restrictions	Syntax
<i>dbslice</i>	Name of the dbslice in which the table is to be stored	The specified dbslice must already exist.	Identifier, p. 4-113
<i>dbspace</i>	Name of the dbspace in which the table is to be stored	The specified dbspace must already exist.	Identifier, p. 4-113

IN Dbospace Clause

The IN *dbospace* clause allows you to isolate a table. The dbospace that you specify must already exist. If you do not specify a location with either the IN *dbospace* clause or a fragmentation scheme, the database server stores the table in the dbospace where the current database resides.

IN Dbslice Clause

If you are using Dynamic Server with AD and XP Options, the *IN dbslice* clause allows you to fragment a table across multiple dbspaces that are defined as a dbslice. The database server fragments the table by round-robin in the dbspaces that make up the dbslice at the time the table is created.

Storage of BYTE and TEXT Columns

In Dynamic Server, you can store data for a BYTE or TEXT column with the table or in a separate blobspace. The following example shows how blobspaces and dbspaces are specified. The user creates the **resume** table. The data for the table is stored in the **employ** dbspace. The data in the **vita** column is stored with the table, but the data associated with the **photo** column is stored in a blobspace named **photo_space**.

```
CREATE TABLE resume
(
  fname          CHAR(15),
  lname          CHAR(15),
  phone          CHAR(18),
  recd_date      DATETIME YEAR TO HOUR,
  contact_date   DATETIME YEAR TO HOUR,
  comments       VARCHAR(250, 100),
  vita           TEXT IN TABLE,
  photo          BYTE IN photo_space
)
IN employ
```

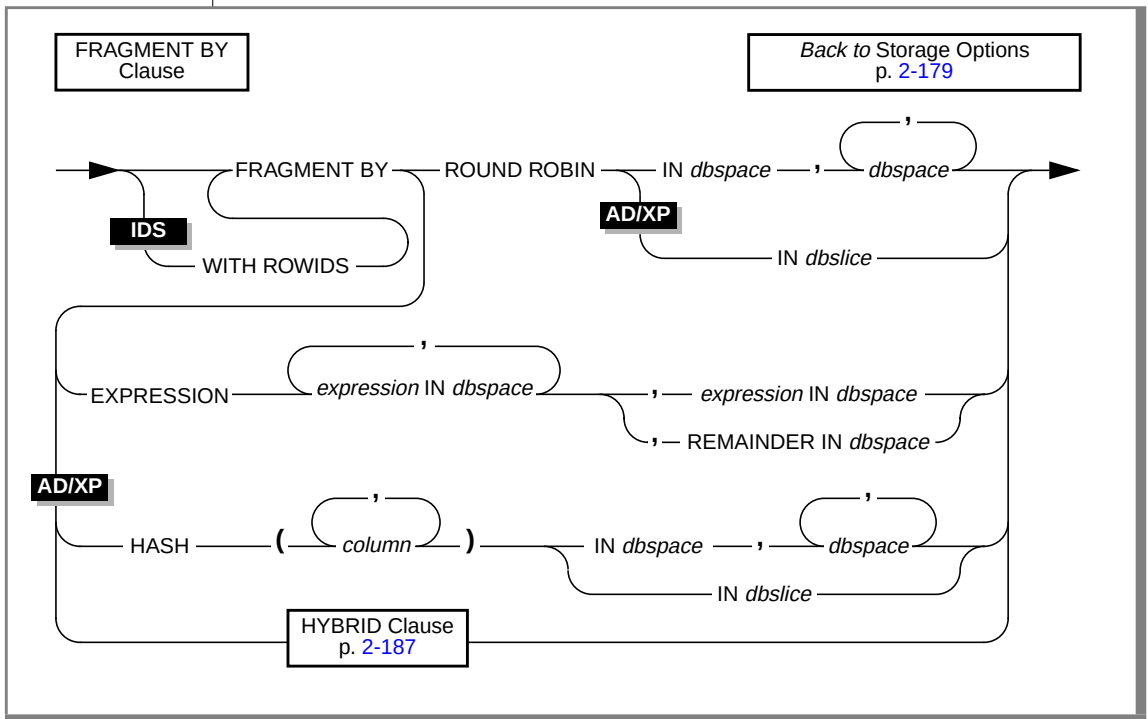
For the syntax of the BYTE or TEXT data types, see the Data Type segment on [page 4-27](#).

W/D

FRAGMENT BY Clause

The FRAGMENT BY clause allows you to create fragmented tables and specify the distribution scheme. Fragmentation means that groups of rows in a table are stored together in the same dbspace.

This clause is not available with Dynamic Server, Workgroup and Developer Editions. ♦



Element	Purpose	Restrictions	Element
<i>column</i>	Name of the column or columns on which you want to fragment your table	All specified columns must be in the current table. If you specify a serial column, you cannot specify any other column.	Identifier, p. 4-113
<i>dbslice</i>	Name of the dbslice that contains the dbspaces (managed on different coservers) in which the table fragments reside	The dbslice must exist when you execute the statement.	Identifier, p. 4-113
<i>dbspace</i>	Name of the dbspace that contains a table fragment If you do not specify a location with either the IN <i>dbspace</i> clause or a fragmentation scheme, the new table is stored in the dbspace where the current table resides.	The dbspaces must exist when you execute the statement. You can specify a maximum of 2,048 dbspaces.	Identifier, p. 4-113
<i>expression</i>	Expression that defines a fragment where a row is to be stored	Each <i>expression</i> can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, serial columns, current date and / or time functions, or aggregates are allowed in a <i>expression</i> .	Expression, p. 4-33

IDS

Using WITH ROWIDS Keywords

In Dynamic Server, 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 keywords 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. Each row contains an additional 4 bytes to store the **rowid** column after you add it.

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



Using the ROUND ROBIN Clause

In a round-robin distribution scheme, the database server distributes the rows among the specified dbspaces 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.

Using the IN Dbslice Clause

In Dynamic Server with AD and XP Options, the IN *dbslice* clause allows you to fragment a table across multiple dbspaces that different coservers manage. When you specify a known dbslice, the database server applies the fragmentation expression to dbspaces that comprise the dbslice.

Using the 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 a location with either the IN *dbspace* clause or a fragmentation scheme, the new table is stored in the dbspace where the current database resides.

Temporary tables do not have a default dbspace. For further information about how to store temporary tables, see the [“TEMP TABLE Clause” on page 2-190](#).

For example, if the **history** database is in the **dbs1** dbspace, but you want the **family** data placed in a separate dbspace called **famdata**, use the following statements:

```
CREATE DATABASE history IN dbs1

CREATE TABLE family
(
    id_num          SERIAL(101),
    name            CHAR(40),
    nickname        CHAR(20),
    mother          CHAR(40),
    father          CHAR(40)
)
IN famdata
```

For more information about how to store your tables in separate dbspaces, see your [Administrator's Guide](#).

Using the *EXPRESSION* Clause

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 dbbsp1,
c1 >= 100 and c1 < 200 IN dbbsp2,
c1 >= 200 IN dbbsp3
```

- 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 dbbsp2,
zip_num = 91120 OR zip_num = 92310 IN dbbsp4,
REMAINDER IN dbbsp5
```



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.”](#)

AD/XP

Using the *HASH* Clause

In Dynamic Server with AD and XP Options, if you use a hash distribution scheme, the database server distributes the rows as you insert them so that the fragments maintain approximately the same number of rows. In this distribution scheme, the database server can eliminate fragments when it searches for a row because the hash is internally known.

For example, if you have a very large database, as in a data-warehousing environment, you can fragment your tables across disks that belong to different coservers. If you expect to perform a lot of queries that scan most of the data, you can use a system-defined hash distribution scheme to balance the I/O processing as follows:

```
CREATE TABLE customer
(
    cust_id integer,
    descr char(45),
    level char(15),
    sale_type char(10),
    channel char(30),
    corp char(45),
    cust char(45),
    vert_mkt char(30),
    state_prov char(20),
    country char(15),
    org_cust_id char(20)
)
FRAGMENT BY HASH (cust_id) IN
    customer1_spc,
    customer2_spc,
    customer3_spc,
    customer4_spc,
    customer5_spc,
    customer6_spc,
    customer7_spc,
    customer8_spc
EXTENT SIZE 16 NEXT SIZE 16
.
.
.
```

This example uses eight coservers with one dbspace defined on each coserver.

Serial Columns in Distribution Scheme of Fragmentation Strategy

In Dynamic Server with AD and XP Options, you can specify a serial column in the hash-distribution scheme of a fragmented table. The following excerpt is a sample SQL CREATE TABLE statement:

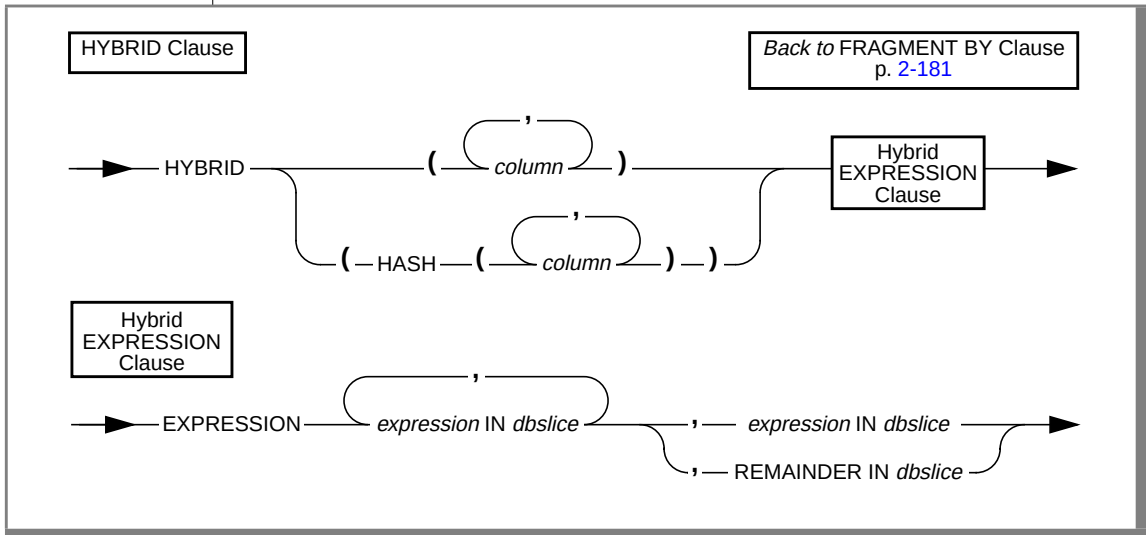
```
CREATE TABLE customer
(
    cust_id serial,
    .
    .
    .
)
FRAGMENT BY HASH (cust_id) IN
    customer1_spc,
    customer2_spc
    .
    .
    .
```

You might notice a difference between serial-column values in fragmented and unfragmented tables. The database server assigns serial values sequentially within fragments, but fragments might contain values from noncontiguous ranges. You cannot specify what these ranges are. The database server controls these ranges and guarantees only that they do not overlap.

AD/XP

Using the HYBRID Clause

In Dynamic Server with AD and XP Options, the HYBRID Clause allows you to apply two distribution schemes to the same table.

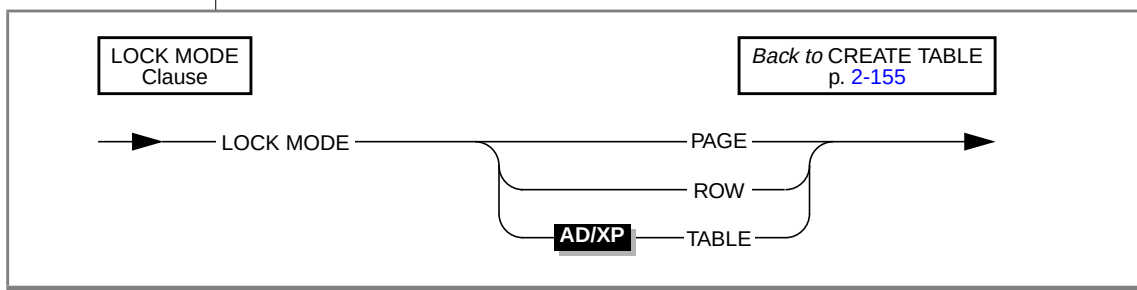


Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of the column or columns on which you want to fragment your table	All specified columns must be in the current table. If you specify a serial column, you cannot specify any other column.	Identifier, p. 4-113
<i>dbslice</i>	Name of the dbslice that contains all of the table fragments	The dbslice must exist when you execute the statement.	Identifier, p. 4-113
<i>expression</i>	Expression that defines a fragment where a row is to be stored	Each <i>expression</i> can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, serial columns, current date and /or time functions, or aggregates are allowed in a <i>expression</i> .	Expression, p. 4-33

Hybrid fragmentation is valid only with the combination of hash and expression distribution schemes. The hash distribution scheme is the default hybrid distribution scheme, so explicitly specifying the HASH clause is optional.

When you specify hybrid fragmentation, the Hybrid-EXPRESSION clause determines the dbslice where the data is stored, and the hash column (or columns) in the HYBRID clause determines the dbspace within the dbslice.

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. You can also exceed the maximum number of locks available, depending on the configuration of your operating system.

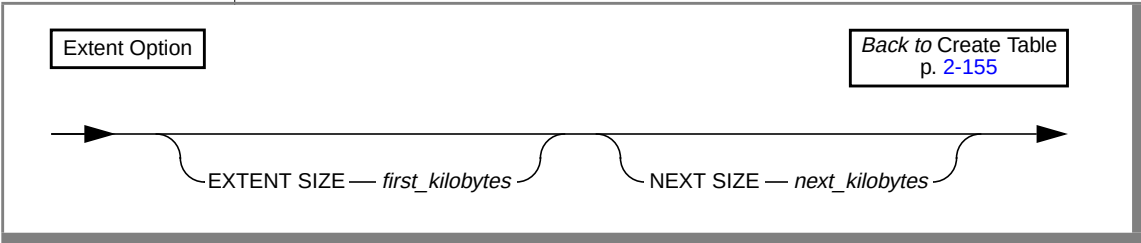
Page locking allows you to obtain and release one lock on a whole page of rows. Page locking is especially useful when you know that the rows are grouped into pages in the same order that you are using to process all the rows. For example, if you are processing the contents of a table in the same order as its cluster index, page locking is especially appropriate. You can change the lock mode of an existing table with the ALTER TABLE statement.

In Dynamic Server with AD and XP Options, table locking provides a lock on an entire table. This type of lock reduces update concurrency in comparison to row and page locks. Multiple read-only transactions can still access the table. A table lock reduces the lock-management overhead for the table. ♦

AD/XP

Extent Option

The extent option defines the size of the extent assigned to the table. If you do not specify the extent size, the default size for all extents is 16 kilobytes.



Element	Purpose	Restrictions	Syntax
<i>first_kilobytes</i>	Length in kilobytes of the first extent for the 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. 4-33
<i>next_kilobytes</i>	Length in kilobytes for the subsequent extents	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. 4-33

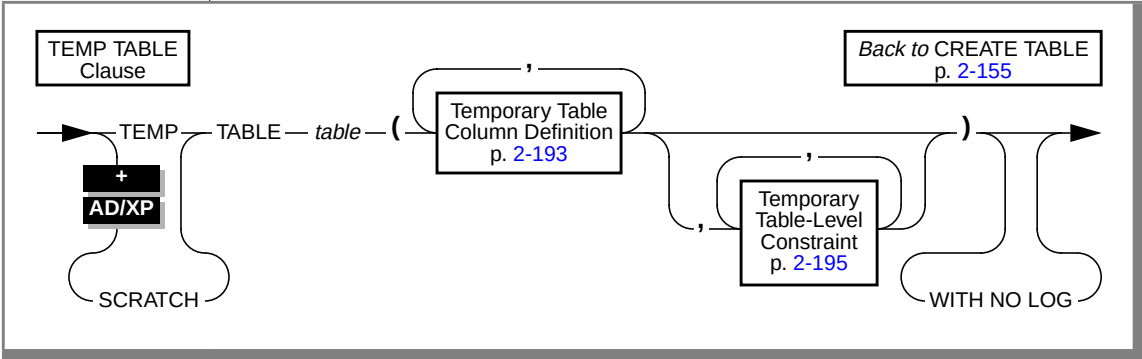
The following example specifies a first extent of 20 kilobytes and allows the rest of the extents to use the default size:

```
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
```

For a discussion about calculating extent sizes, see your [Performance Guide](#).

TEMP TABLE Clause

The TEMP TABLE clause allows you to create temporary tables. Temporary tables have many of the same features as permanent tables, but the tables are not preserved when you exit from the database. Users are not allowed to alter temporary tables in the same way that they can alter permanent tables.



Element	Purpose	Restrictions	Syntax
table	Name that you want to assign to the temporary table	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.	Database Object Name, p. 4-25

Temporary tables are visible to the session creating those tables. They are not visible to other users or sessions. Temporary tables do not appear in the system catalogs.

A temporary table is associated with a session, not a database. Therefore, when you create a temporary table, you cannot create another temporary table with the same name (even for another database) until you drop the first temporary table or end the session.

AD/XP

Using the *SCRATCH* Keyword

In Dynamic Server with AD and XP Options, a *SCRATCH* temporary table is a non-logging temporary table that cannot contain index or referential constraints. A *SCRATCH* table is identical to a *TEMP* table created with the *WITH NO LOG* option.

Using the *WITH NO LOG* Keywords

Use the *WITH NO LOG* keywords to prevent logging of temporary tables in databases started with logging. You must use the *WITH NO LOG* keywords on temporary tables created in temporary dbspaces.

IDS

In Dynamic Server, if you use the *WITH NO LOG* keywords in a database that 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
```

Explicit Temporary Tables

Temporary tables that you create are called *explicit* temporary tables. You can create explicit temporary tables in the following ways:

- With the *CREATE TEMP TABLE* statement
- With the *SELECT...INTO TEMP* statement.

If you have the *Connect* privilege on a database, you can create explicit temporary tables. Once a temporary table is created, you can build indexes on the table. However, you are the only user who can see the temporary table.

AD/XP

If you are using Dynamic Server with AD and XP Options, you can fragment an explicit temporary table across disks that belong to different coservers. ♦

Duration of Temporary Tables

When an application creates an explicit temporary table, it exists until one of the following situations occurs:

- The application terminates.
- A DROP TABLE statement is issued.
- 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.

Temporary Tables and DB-Access

In DB-Access, you cannot use the INFO statement and the Info Menu Option with temporary tables. ♦

Names of 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.

Location of Temporary Tables

You can specify where temporary tables are created with the CREATE TEMP TABLE statement, environment variables, and ONCONFIG parameters. The database server looks at the following information, in order, to find where to store temporary tables:

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

If you do not use the *IN dbspace* clause or the *FRAGMENT BY* clause to fragment the table, the database server checks to see if the **DBSPACETEMP** environment variable is set. The **DBSPACETEMP** environment variable specifies where temporary tables can be stored. If the environment variable is set, the database server stores the temporary table in one of the storage spaces specified in that list.

4. The ONCONFIG parameter DBSPACETEMP

You can specify a location for temporary tables with the ONCONFIG parameter DBSPACETEMP.

If you do not use the *IN dbspace* clause, the *FRAGMENT BY* clause to fragment the table, the **DBSPACETEMP** environment variable, or the ONCONFIG parameter DBSPACETEMP, the temporary tables are created in the same dbspace as your database server. ♦

Specifying Multiple dbspaces for Temporary Tables

You can specify more than one dbspace for the **DBSPACETEMP** environment variable. The following examples show how you can specify the dbspace definitions for the **DBSPACETEMP** environment variable:

UNIX

```
setenv DBSPACETEMP tempspc1:tempspc2:tempspc3
```

♦

WIN NT

```
set DBSPACETEMP=tempspc1:tempspc2:tempspc3
```

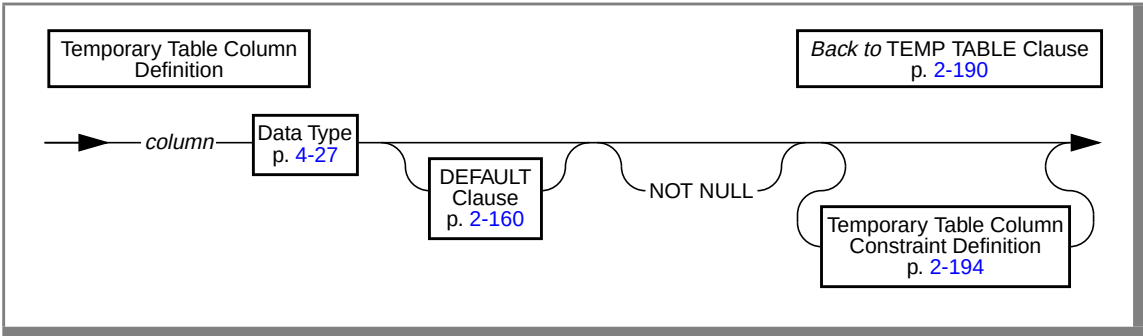
♦

Each temporary table that you create round-robins to a dbspace. For example, if you create three temporary tables, the first one goes into the dbspace called **tempspc1**, the second one goes into **tempspc2**, and the third one goes into **tempspc3**.

For additional information about the **DBSPACETEMP** environment variable, refer to the [Informix Guide to SQL: Reference](#). For additional information about the ONCONFIG parameter DBSPACETEMP, see your [Administrator's Guide](#).

Temporary-Table Column Definition

CREATE TABLE

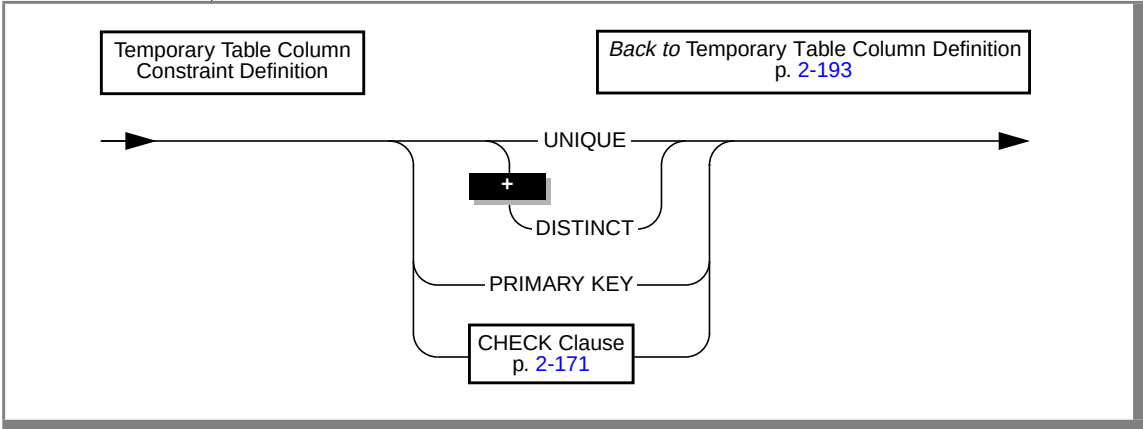


Element	Purpose	Restrictions	Syntax
column	Name of a column in the table	Name must be unique within a table, but you can use the same names in different tables in the same database.	Identifier, p. 4-113

You define columns for temporary tables in the same manner as you define columns for regular database tables. The only difference is the option for defining column constraints, which is defined in the following section.

For more information about how to define single columns for temporary tables, see the [“Column Definition” on page 2-159](#).

Temporary Table Column Constraint Definition

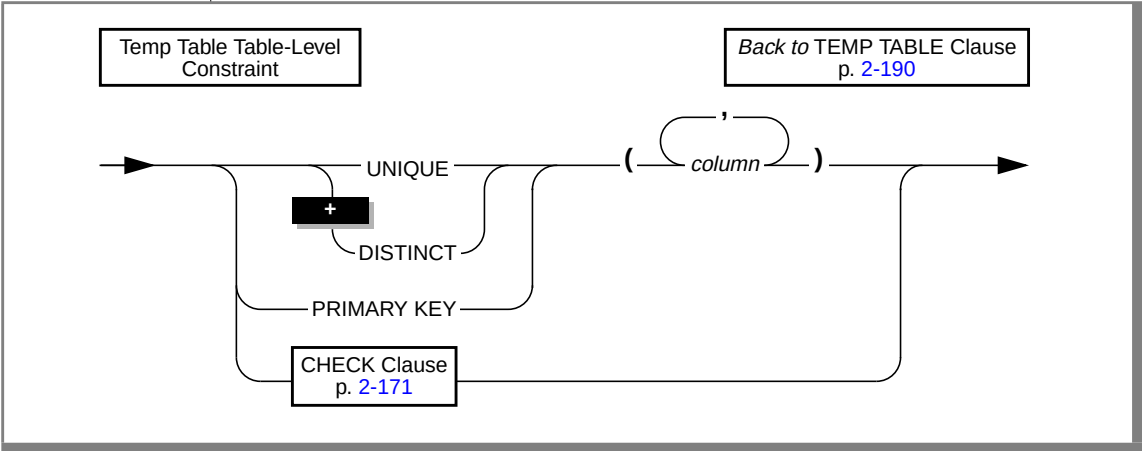


Temporary-table column constraints are the same as column constraints for regular tables, with the following exceptions:

- You cannot place referential constraints on columns in a temporary table.
- You cannot assign a name to a constraint on a temporary-table column.
- You cannot set the database object mode of a constraint on a temporary-table column.

For information about column constraints in regular tables, see [“Column-Level Constraints” on page 2-163](#).

Table-Level Constraint for Temporary Tables



Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of the column or columns on which the constraint is placed	See “Restrictions on Table-Level Constraints for Temporary Tables” on page 2-196 .	Identifier, p. 4-113

You can place a table-level constraint on one or more columns of a temporary table.

Restrictions on Table-Level Constraints for Temporary Tables

Table-level constraints are defined for temporary tables in the same manner as for regular database tables, with the following exceptions:

- You cannot place referential constraints on columns in a temporary table.
- You cannot assign a name to a constraint on a temporary-table column.
- You cannot set the database object mode of a constraint on a temporary-table column.

For information about table-level constraints on regular tables, see [“Table-Level Constraints” on page 2-175](#).

References

Related statements: ALTER TABLE, CREATE INDEX, CREATE DATABASE, DROP TABLE, and SET DATABASE OBJECT MODE

For discussions of data-integrity constraints and database and table creation, see the [Informix Guide to Database Design and Implementation](#).

For a discussion of the ON DELETE CASCADE clause, see the [Informix Guide to SQL: Tutorial](#).

For a discussion of extent sizing, see your [Performance Guide](#).

+

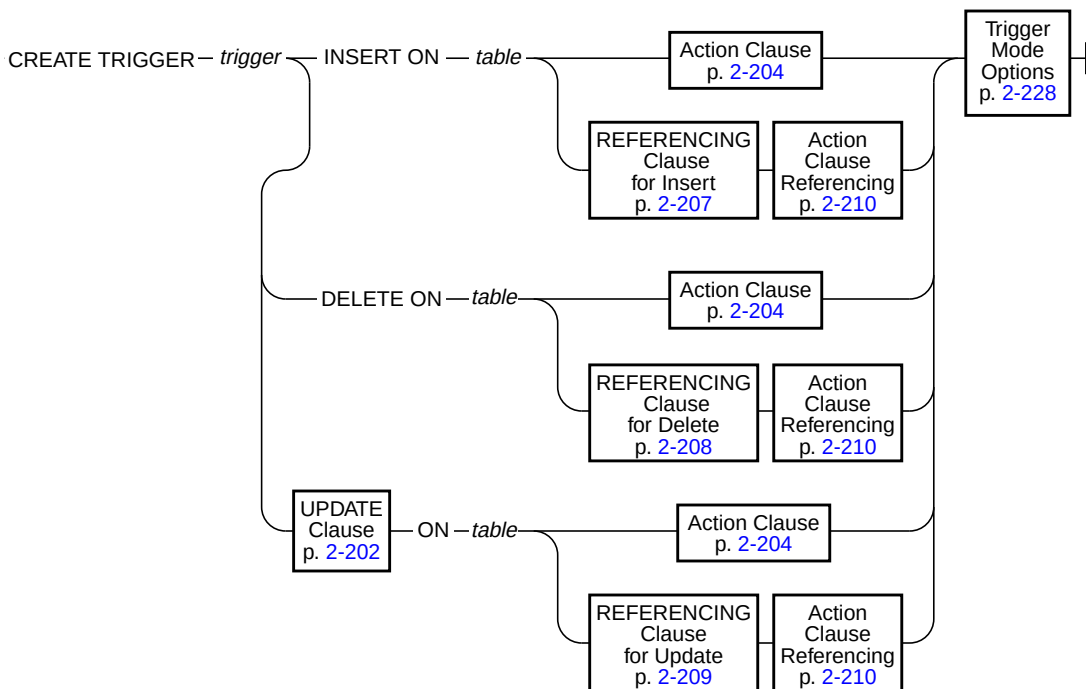
IDS

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.

You can use this statement only with Dynamic Server.

Syntax



Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of the table that the trigger affects	The name must be different from any existing table, view, or synonym name in the current database.	Database Object Name, p. 4-25
<i>trigger</i>	Name of the trigger	You can specify a trigger for the current database only. The name of the trigger must be unique.	Database Object Name, p. 4-25

Usage

You must be either the owner of the table or have the 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.

In DB-Access, if you want to define a trigger as part of a schema, place the CREATE TRIGGER statement inside a CREATE SCHEMA statement. ♦

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
```

For a list of data manipulation statements, see “[Data Manipulation Statements](#)” on page 1-10.

You cannot use a stored procedure variable in a CREATE TRIGGER statement.

If you are embedding the CREATE TRIGGER statement in an ESQL/C program, you cannot use a host variable in the trigger specification. ♦

DB

E/C

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, or more information about multiple triggers on the same table. For more information about multiple triggers on the same table, see [“UPDATE Clause” on page 2-202](#).

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. For more information on the behavior of triggered actions, see [“Action Clause” on page 2-204](#) and [“Triggered Action List” on page 2-211](#).

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 2-204](#).

Privileges on the Trigger Event

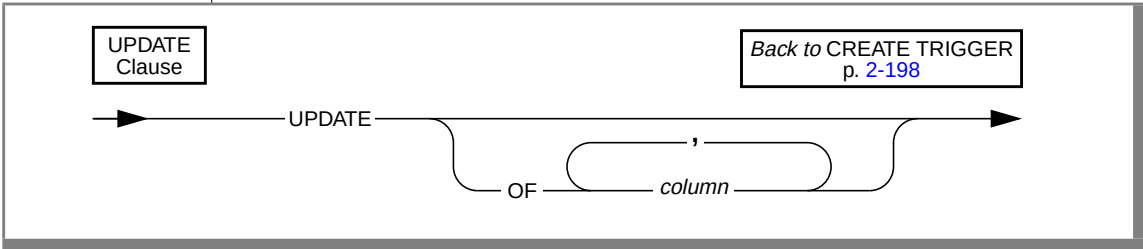
You must have the appropriate Insert, Delete, or Update privilege on the triggering table to execute the INSERT, DELETE, or UPDATE statement that is the trigger event. The triggering statement might still fail, however, if you do not have the privileges necessary to execute one of the SQL statements 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 2-222](#).

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 2-224](#).

UPDATE Clause



Element	Purpose	Restrictions	Syntax
<i>column</i>	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. 4-113

If the trigger event is an UPDATE statement, the trigger executes when any column in the triggering column list is updated.

If the trigger event is an UPDATE statement and you do not specify the OF *column* option in the definition of the trigger event, the trigger executes when any column in the triggering table 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 shown in the following example:

```
CREATE TRIGGER trig1 UPDATE OF a, c ON taba
AFTER (UPDATE tabb SET y = y + 1);

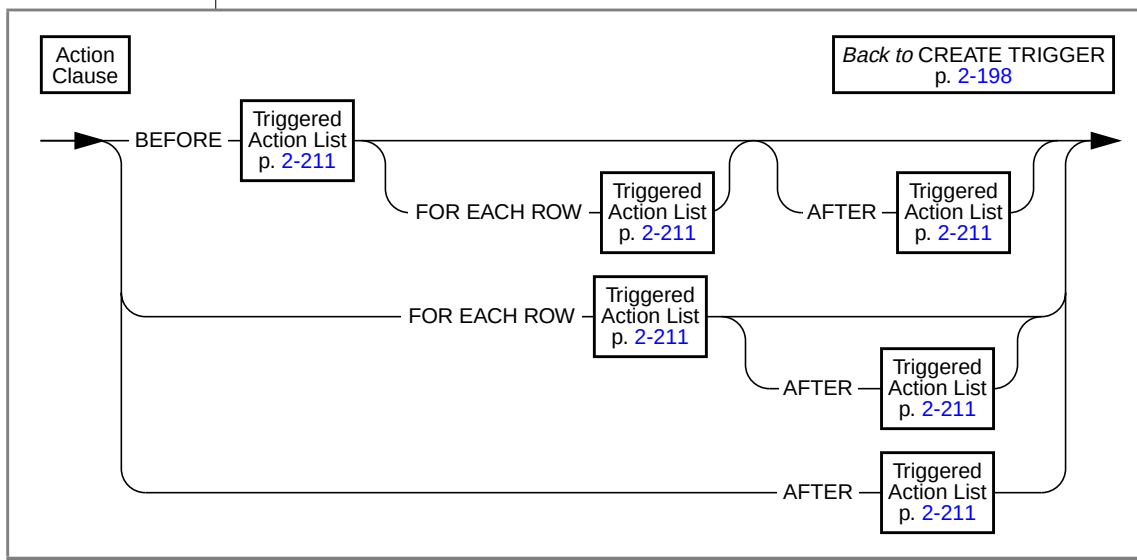
CREATE TRIGGER trig2 UPDATE OF b, d ON taba
AFTER (UPDATE tabb SET z = z + 1);
```

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



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. For more information on the action clause when a **REFERENCING** clause is present, see [“Action Clause Referencing” on page 2-210](#).

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 shown in the following example:

```
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, then 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.

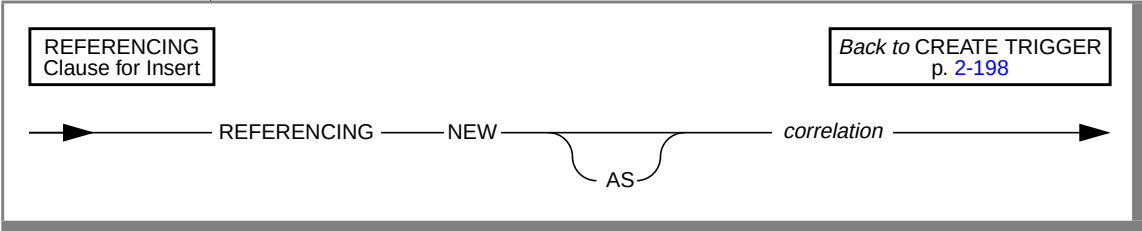
Guaranteeing Row-Order Independence

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 2-224](#).
- 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.

REFERENCING Clause for Insert



Element	Purpose	Restrictions	Syntax
correlation	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. 4-113

Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See [“Action Clause Referencing” on page 2-210](#).

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 how to use correlation names, see [“Using Correlation Names in Triggered Actions” on page 2-214](#).

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**.

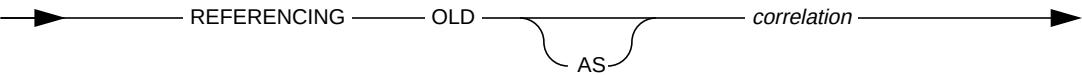
```
CREATE TABLE table1 (col1 INT, col2 INT);
CREATE TABLE backup_table1 (col1 INT, col2 INT);
CREATE TRIGGER before_trig
  INSERT ON table1
  REFERENCING NEW as new
  FOR EACH ROW
  (
    INSERT INTO backup_table1 (col1, col2)
    VALUES (new.col1, new.col2)
  );
```

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.

REFERENCING Clause for Delete

REFERENCING
Clause for Delete

Back to CREATE TRIGGER
p. 2-198



Element	Purpose	Restrictions	Syntax
correlation	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. 4-113

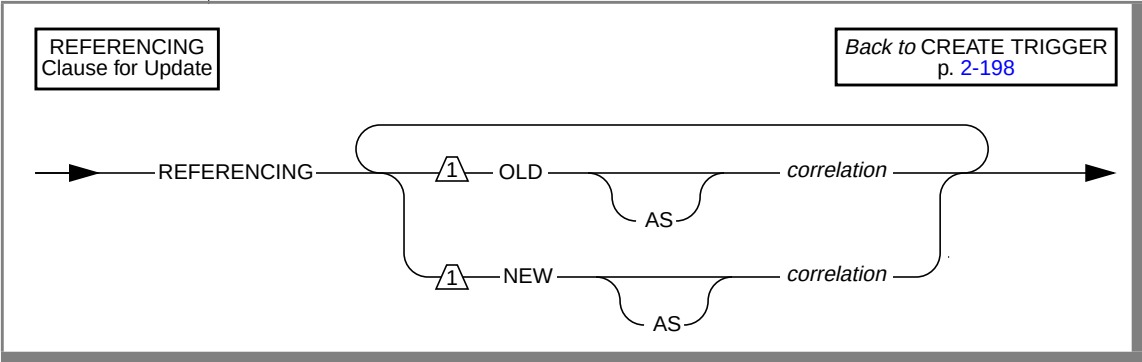
Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See [“Action Clause Referencing” on page 2-210](#).

You 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. For the rules governing the use of correlation names, see [“Using Correlation Names in Triggered Actions” on page 2-214](#).

You can use the DELETE REFERENCING clause only if you define a FOR EACH ROW triggered action.

REFERENCING Clause for Update



Element	Purpose	Restrictions	Syntax
correlation	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 executing the triggering statement.	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. 4-113

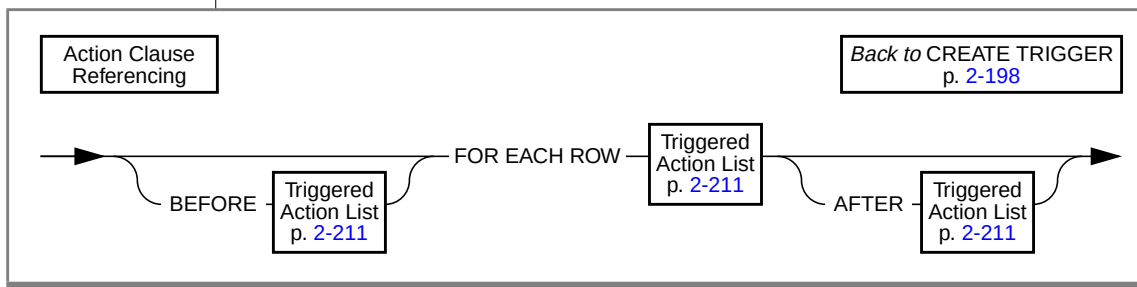
Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See [“Action Clause Referencing” on page 2-210](#).

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. For the rules that govern the use of correlation names, see [“Using Correlation Names in Triggered Actions” on page 2-214](#).

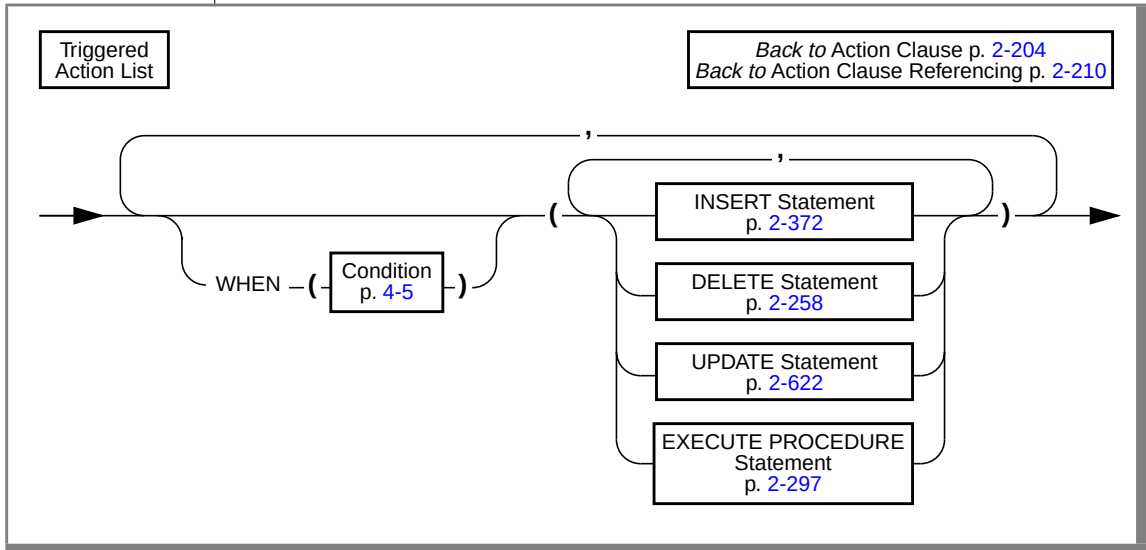
You can use the UPDATE REFERENCING clause only if you define a FOR EACH ROW triggered action.

Action Clause Referencing



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. For information on the BEFORE, FOR EACH ROW, and AFTER triggered-action lists, see [“Action Clause” on page 2-204](#).

Triggered Action List



The triggered action consists of an optional WHEN condition and the action statements. Database 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 database objects that are referenced directly in the trigger definition.

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.

CREATE TRIGGER

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 stored procedure that executes inside the WHEN condition carries the same restrictions as a stored procedure that is called in a data manipulation statement. For more information about a stored procedure that is called within a data manipulation statement, see the CREATE PROCEDURE statement on [page 2-134](#).

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 stored procedure 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 non-qualified column name.

For example, assume that the following DELETE statement is a triggered action inside the FOR EACH ROW section of a trigger:

```
DELETE FROM tab1 WHERE col_c = col_c2
```

For the statement to be valid, both **col_c** and **col_c2** must be columns from **tab1**. If **col_c2** is intended to be a correlation reference to a column in the triggering table, it must be qualified by either the old or the new correlation name. If **col_c2** is not a column in **tab1** and is not qualified by either the old or new correlation name, you get an error.

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	The value before the triggering statement.
Current Value	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.

Reentrancy of Triggers

In some cases a trigger can be reentrant. In these cases the triggered action can reference the triggering table. In other words, both the trigger event and the triggered action can operate on the same table. The following list summarizes the situations in which triggers can be reentrant and the situations in which triggers cannot be reentrant:

- If the trigger event is an UPDATE statement, the triggered action cannot be an INSERT or DELETE statement that references the table that was updated by the trigger event.
- If the trigger event is an UPDATE statement, the triggered action cannot be an UPDATE statement that references a column that was updated by the trigger event.

However, if the trigger event is an UPDATE statement, and the triggered action is also an UPDATE statement, the triggered action can update a column that was not updated by the trigger event.

For example, assume that the following UPDATE statement, which updates columns **a** and **b** of **tab1**, is the triggering statement:

```
UPDATE tab1 SET (a, b) = (a + 1, b + 1)
```

Now consider the 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
```


- If the trigger event is an UPDATE statement, the triggered action can be an EXECUTE PROCEDURE statement with an INTO clause that references a column that was updated by the trigger event or any other column in the triggering table.

When an EXECUTE PROCEDURE statement is the triggered action, you can specify the INTO clause for an UPDATE trigger only 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)
```

The column that follows the INTO keyword can be a column in the triggering table that was updated by the trigger event, or a column in the triggering table that was not updated by the trigger event.

When the INTO clause appears in the EXECUTE PROCEDURE statement, the database server updates the columns named there with the values returned from the stored procedure. The database server performs the update immediately upon returning from the stored procedure.

- If the trigger event is an INSERT statement, the triggered action cannot be an INSERT or DELETE statement that references the triggering table.

- If the trigger event is an INSERT statement, the triggered action can be an UPDATE statement that references a column in the triggering table. However, this column cannot be a column for which a value was supplied by the trigger event.

If the trigger event is an INSERT, and the triggered action is an UPDATE on the triggering table, the columns in both statements must be mutually exclusive. For example, assume that the trigger event is an INSERT statement that inserts values for columns **cola** and **colb** of table **tab1**:

```
INSERT INTO tab1 (cola, colb) VALUES (1,10)
```

Now consider the triggered actions. The first UPDATE statement is valid, but the second one is not because it updates column **colb** even though the trigger event already supplied a value for column **colb**.

```
UPDATE tab1 SET colc=100; --OK
UPDATE tab1 SET colb=100; --ILLEGAL
```

- If the trigger event is an INSERT statement, the triggered action can be an EXECUTE PROCEDURE statement with an INTO clause that references a column that was supplied by the trigger event or a column that was not supplied by the trigger event.

When an EXECUTE PROCEDURE statement is the triggered action, you can specify the INTO clause for an INSERT trigger only 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 ins_totpr INSERT ON items
REFERENCING NEW as new_ins
FOR EACH ROW (EXECUTE PROCEDURE
    calc_totpr(0, new_ins.quantity, 0)
    INTO total_price).
```

The column that follows the INTO keyword can be a column in the triggering table that was supplied by the trigger event, or a column in the triggering table that was not supplied by the trigger event.

When the INTO clause appears in the EXECUTE PROCEDURE statement, the database server updates the columns named there with the values returned from the stored procedure. The database server performs the update immediately upon returning from the stored procedure.

- If the triggered action is a SELECT statement, the SELECT statement can reference the triggering table. 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.

Reentrancy and Cascading Triggers

The cases when a trigger cannot be reentrant apply recursively to all cascading triggers, which are considered part of the initial trigger. In particular, this rule means that a cascading trigger cannot update any columns in the triggering table that were updated by the original triggering statement, including any nontriggering columns affected by that statement. For example, assume 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.

```
CREATE TRIGGER trig1 UPDATE OF a ON tab1-- Valid
  AFTER (UPDATE tab2 set e = e + 1);

CREATE TRIGGER trig2 UPDATE of e ON tab2-- Invalid
  AFTER (UPDATE tab1 set b = b + 1);
```

Now consider the following SQL statements. When the final UPDATE statement is executed, column **a** is updated and the trigger **trig1** is activated. The triggered action again updates column **a** with an EXECUTE PROCEDURE INTO statement.

```
create table temp (a int, b int, e int);
insert into temp values (10, 20, 30);

create procedure proc(val int)
returning int,int;
return val+10, val+20;
end procedure;

create trigger trig1 update of a on temp
for each row (execute procedure proc(50) into a, e);

create trigger trig2 update of e on temp
for each row (execute procedure proc(100) into a, e);

update temp set (a,b) = (40,50);
```

Several questions arise from this example of cascading triggers. First, should the update of column **a** activate trigger **trig1** again? The answer is no. Because the trigger has been activated, it is stopped from being activated a second time. Whenever the triggered action is an EXECUTE PROCEDURE INTO statement, the only triggers that are activated are those that are defined on columns that are mutually exclusive from the columns in that table updated until then (in the cascade of triggers). Other triggers are ignored.

Another question that arises from the example is whether trigger **trig2** should be activated. The answer is yes. The trigger **trig2** is defined on column **e**. Until now, column **e** in table **temp** has not been modified. Trigger **trig2** is activated.

A final question that arises from the example is whether triggers **trig1** and **trig2** should be activated after the triggered action in **trig2** is performed. The answer is no. Neither trigger is activated. By this time columns **a** and **e** have been updated once, and triggers **trig1** and **trig2** have been executed once. The database server ignores these triggers instead of firing them.

For more information about cascading triggers, see [“Cascading Triggers” on page 2-224](#).

Rules for Stored Procedures

In addition to the rules listed in [“Reentrancy of Triggers” on page 2-217](#), the following rules apply to a stored procedure that is used as a triggered action:

- The stored procedure cannot be a cursory procedure (that is, a procedure that returns more than one row) in a place where only one row is expected.
- You cannot use the old or new correlation name inside the stored procedure. If you need to use the corresponding values in the procedure, you must pass them as parameters. The stored procedure 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 stored procedure: 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 stored procedure as a triggered action, the database 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 a stored procedure, you must have the Execute privilege on the procedure or the owner of the trigger must have the Execute privilege and the WITH GRANT OPTION privilege. Inside the stored procedure, you do not carry the privileges of the trigger owner; instead you have the following privileges:

1. The triggered action is a DBA-privileged procedure.

When you are granted the Execute privilege on the procedure, the database server automatically grants you DBA privileges for the procedure execution. These DBA privileges are available only when you are executing the procedure.

2. The triggered action is an owner-privileged procedure.

If the procedure owner has the WITH GRANT OPTION right for the necessary privileges on the underlying database objects, you inherit these privilege when you are granted the Execute privilege. In this case, all the non-qualified database objects that the procedure references are qualified by the name of the procedure owner.

If the procedure owner does not have the WITH GRANT OPTION right, you have your original privileges on the underlying database objects when the procedure executes.

For more information on privileges on stored procedures, refer to 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.

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 logging, however, the triggers execute successfully because constraint checking is deferred until all the triggered actions are complete, including the actions of cascading triggers. For more information about how constraints are handled when triggers execute, see [“Constraint Checking” on page 2-225](#).

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, or an INSERT statement. INSERT trigger statements can have UPDATE triggered actions on the same table.

Constraint Checking

When you use logging, the database server defers constraint checking on the triggering statement until after the statements in the triggered-action list execute. The database server effectively executes a SET CONSTRAINTS ALL DEFERRED statement before it executes the triggering statement. After the triggered action is completed, it effectively executes a SET CONSTRAINTS *constraint* IMMEDIATE statement to check the constraints that were deferred. This action allows you to write triggers so that the triggered action can resolve any constraint violations that the triggering statement creates. For more information, see the SET Database Object Mode statement on [page 2-512](#).

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, the database 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.

You cannot use the SET statement in a triggered action. The database 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

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.

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
```

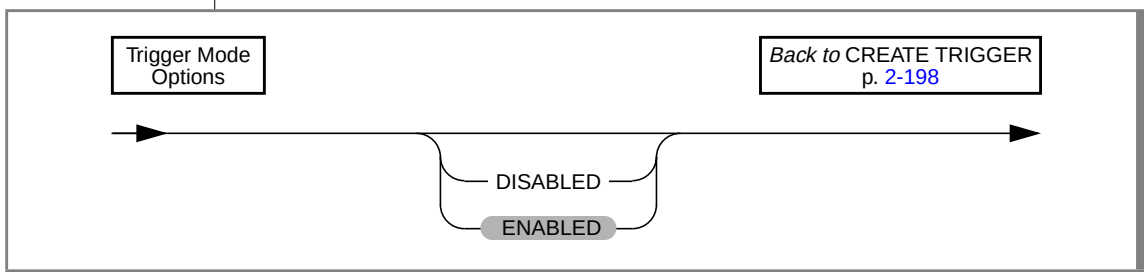
When the RAISE EXCEPTION statement returns the error, however, the database server rolls back this insert because it is part of the triggered actions. If the procedure is executed outside a triggered action, the insert is not rolled back.

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 Mode Options

Use the trigger-mode option to enable or disable a trigger when you create it.



You can create triggers in the following modes.

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 Modes for Triggers

You must observe the following rules when you specify the mode for a trigger in the CREATE TRIGGER statement:

- If you do not specify a mode, the trigger is enabled by default.
- You can use the SET Database Object Mode 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.
- You cannot create a trigger on a violations table or a diagnostics table.

References

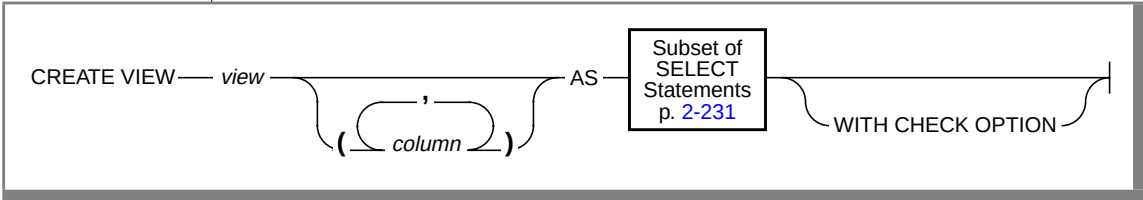
Related statements: DROP TRIGGER, CREATE PROCEDURE, EXECUTE PROCEDURE, and SET DATABASE OBJECT MODE

For a task-oriented discussion of triggers, see the [Informix Guide to SQL: Tutorial](#).

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
<i>column</i>	Name of a column in the view	See “Naming View Columns” on page 2-232 .	Identifier, p. 4-113
<i>view</i>	Name of the view	The name must be unique in the database.	Database Object Name, p. 4-25

Usage

Except for the statements in the following list, you can use a view in any SQL statement where you can use a table.

ALTER FRAGMENT	DROP INDEX
ALTER INDEX	DROP TABLE
ALTER TABLE	DROP TRIGGER
CREATE INDEX	LOCK TABLE
CREATE TABLE	RENAME TABLE
CREATE TRIGGER	UNLOCK TABLE

The view behaves like a table that is called *view*. 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 in the tables from which the view is derived. The database server determines data types of virtual columns 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.

In DB-Access, if you create a view outside the CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

Subset of SELECT Statements Allowed in CREATE VIEW

The SELECT statement has the form that is described on [page 2-450](#), but in CREATE VIEW, the FROM clause of the SELECT statement cannot contain the name of a temporary table.

Do not use display labels in the select list. Display labels in the select list are interpreted as column names.

The SELECT statement in CREATE VIEW cannot include the following clauses:

- ★ FIRST
- INTO TEMP
- ORDER BY

Union Views

The SELECT statement in CREATE VIEW can contain a UNION or UNION ALL operator. A view that contains a UNION or UNION ALL operator in its SELECT statement is known as a union view. Observe the following restrictions on union views:

- If a CREATE VIEW statement defines a union view, you cannot specify the WITH CHECK OPTION keywords in the CREATE VIEW statement.
- All restrictions that apply to UNION or UNION ALL operations in standalone SELECT statements also apply to UNION and UNION ALL operations in the SELECT statement of a union view. For a list of these restrictions, see [“Restrictions on a Combined SELECT” on page 2-500](#).

For an example of a CREATE VIEW statement that defines a union view, see [“Naming View Columns” on page 2-232](#).

Naming View Columns

The number of columns that you specify in the *column* 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. In the following example, the user must specify the *column* parameter because the select list of the SELECT statement contains an aggregate expression.

```
CREATE VIEW newview (firstcol, secondcol) AS
  SELECT sum(col_a), col_b
  FROM oldtab
```

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
```

You must also provide column names in the *column* parameter when the SELECT statement includes a UNION or UNION ALL operator and the names of the corresponding columns in the SELECT statements are not identical. In the following example, the user must specify the *column* parameter since the second column in the first SELECT statement has a different name from the second column in the second SELECT statement.

```
CREATE VIEW myview (cola, colb) AS
  SELECT colx, coly from firsttab
 UNION
  SELECT colx, colz from secondtab
```

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 discussed in the [Informix Guide to Database Design and Implementation](#). 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 with an arithmetical expression
- A UNION operator

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.*

References

Related statements: CREATE TABLE, DROP VIEW, GRANT, SELECT, and SET SESSION AUTHORIZATION

For a discussion of views, see the [Informix Guide to Database Design and Implementation](#).

+

DATABASE

Use the DATABASE statement to select an accessible database as the current database.

Syntax



Element	Purpose	Restrictions	Syntax
<i>database</i>	Name of the database to select	The database must exist.	Database Name, p. 4-22

Usage

You can use the DATABASE statement to select any database on your database server. To select a database on another database server, specify the name of the database server with the database name.

If you 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.

E/C

Using the DATABASE Statement with ESQL/C

In ESQL/C, you cannot include the DATABASE statement in a multistatement PREPARE operation.

You can determine the characteristics of a 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 (the **sqlca.sqlwarn.sqlwarn1** field) contains a W after the DATABASE statement executes.

ANSI

In an ANSI-compliant database, the third element of the **sqlwarn** structure (the **sqlca.sqlwarn.sqlwarn2** field) contains a W after the DATABASE statement executes. ♦

IDS

If you are using Dynamic Server, the fourth element of the **sqlwarn** structure (the **sqlca.sqlwarn.sqlwarn3** field) contains a W after the DATABASE statement executes.

If the database is running in secondary mode, the seventh element of the **sqlwarn** structure (the **sqlca.sqlwarn.sqlwarn6** field) 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

Related statements: CLOSE DATABASE and CONNECT

For discussions of how to use different data models to design and implement a database, see the [Informix Guide to Database Design and Implementation](#).

+

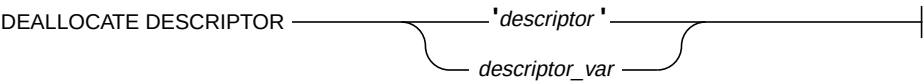
E/C

DEALLOCATE DESCRIPTOR

Use the DEALLOCATE DESCRIPTOR statement to free a previously allocated, system-descriptor area.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>descriptor</i>	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated. The surrounding quotes must be single.	Quoted String, p. 4-157
<i>descriptor_var</i>	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_var* identifies. It also frees all the value descriptors (including memory for data values in the value descriptors).

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 **sqli** structure. You can use it only to free the memory that is allocated for a system-descriptor area.

The following examples show valid DEALLOCATE DESCRIPTOR statements. The first line uses an embedded-variable name, and the second line uses a quoted string to identify the allocated system-descriptor area.

```
EXEC SQL deallocate descriptor :descname;  
EXEC SQL deallocate descriptor 'desc1';
```

References

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

For more information on system-descriptor areas, refer to the [*INFORMIX-ESQL/C Programmer's Manual*](#).

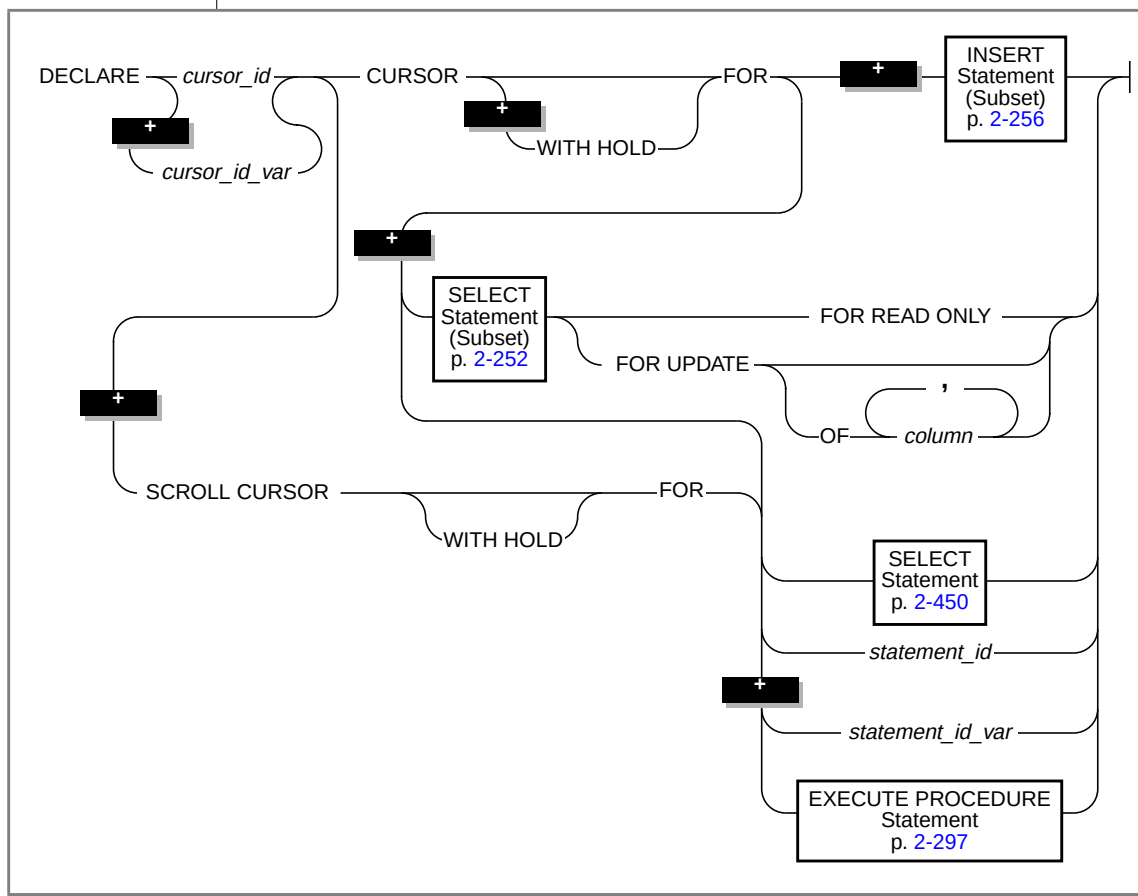
E/C

DECLARE

Use the DECLARE statement to define a cursor that represents the active set of rows that a SELECT, INSERT, or EXECUTE PROCEDURE statement specifies.

Use this statement with ESQL / C.

Syntax



DECLARE

Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of the column to 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. 4-113
<i>cursor_id</i>	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. 4-113
<i>cursor_id_var</i>	Embedded variable that holds the value of <i>cursor_id</i>	Variable must be a character data type.	Name must conform to language-specific rules for variable names.
<i>statement_id</i>	Statement identifier that is a data structure that represents the text of a prepared statement	The <i>statement_id</i> must have already been specified in a PREPARE statement in the same program.	Identifier, p. 4-113
<i>statement_id_var</i>	Embedded variable that holds the value of <i>statement_id</i>	Variable must be a character data type.	Name must conform to language-specific rules for variable names.

Usage

The DECLARE statement associates the cursor with a SELECT, INSERT, or EXECUTE PROCEDURE statement or with the statement identifier (*statement_id* or *statement_id_var*) of a prepared statement.

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.

When the cursor is used with a SELECT statement, it is a data structure that represents a specific location within the active set of rows that the SELECT statement retrieved. You associate a cursor with an INSERT statement if you want to add multiple rows to the database in an INSERT operation. When the cursor is used with an INSERT statement, it represents the rows that the INSERT statement is to add to the database. When the cursor is used with an EXECUTE PROCEDURE statement, it represents the columns or values that the stored procedure retrieved.

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_var` to release the resources that a prepared statement holds; use `FREE cursor_id` or `FREE cursor_id_var` 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.

A variable that you use in place of the cursor name or statement identifier must be the `CHARACTER` data type. In `ESQL/C` programs, the variable must be defined as `exec sql char`.

To declare multiple cursors, use a single statement identifier. For instance, the following `ESQL/C` example does not return an error:

```
EXEC SQL prepare id1 from 'select * from customer';
EXEC SQL declare x cursor for id1;
EXEC SQL declare y scroll cursor for id1;
EXEC SQL declare z cursor with hold for id1;
```

If you include the **-ansi** compilation flag (or if **DBANSIWARN** is set), warnings are generated for statements that use dynamic cursor names or dynamic statement identifier names. 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. The following example uses a host variable to hold the cursor name.

```
EXEC SQL declare x cursor for
    select * from customer;
. . .
strcpy("x", s);
EXEC SQL declare :s cursor for
    select * from customer;
```



Overview of Cursor Types

Functionally, a cursor can be associated with an EXECUTE PROCEDURE statement (a *procedure cursor*) that returns values, an INSERT statement (an *insert cursor*) or a SELECT statement (a *select cursor*). You can use a select cursor to update or delete rows; it is called an *update cursor*.

A cursor can also be associated with a statement identifier, which enables you to use the cursor with EXECUTE PROCEDURE, INSERT, or SELECT statements that are 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.

Tip: *Cursors for stored procedures behave the same as select cursors that are enabled as update cursors.*

Select Cursor or Procedure Cursor

A select or procedure cursor enables you to scan multiple rows of data and to move data row by row into a set of receiving variables, as the following steps describe:

1. DECLARE

Use a DECLARE statement to define a cursor for the SELECT statement or for the EXECUTE PROCEDURE statement.

2. OPEN

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. FETCH

Retrieve successive rows of data with the FETCH statement.

4. CLOSE

Close the cursor with the CLOSE statement when the active set is no longer needed.

5. FREE

Free the cursor with the FREE statement. The FREE statement releases the resources that are allocated for a declared cursor.

A select cursor can be explicitly declared as read only with the FOR READ ONLY option.

Read-Only Cursor

Use the FOR READ ONLY option to state explicitly that a select cursor cannot be used to modify data. In a database that is not ANSI compliant, a select cursor and a select cursor that is built with the FOR READ ONLY option are the same. Neither can be used to update data.

ANSI

In an ANSI-compliant database, if you want a select cursor to be read only, you must use the FOR READ ONLY keywords when you declare the cursor. ♦

Update Cursor

Use the FOR UPDATE option to declare an update cursor. You can use the update cursor to modify (update or delete) the current row.

ANSI

In an ANSI-compliant database, you can use a select cursor to update or delete data as long as the cursor was not declared with the FOR READ ONLY keywords and it follows the restrictions on update cursors that are described in [“Subset of the SELECT Statement Associated with Cursors” on page 2-252](#). You do not need to use the FOR UPDATE keywords when you declare the cursor. ♦

Insert Cursor

An insert cursor increases processing efficiency (compared with embedding the INSERT statement directly). The insert cursor allows bulk insert data to be buffered in memory and written to disk when the buffer is full. 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 as a *sequential* cursor (the default condition), a *scroll* cursor (using the SCROLL keyword), or a *hold* cursor (using the WITH HOLD keywords). 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, 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 ESQ/C example is read only in a database that is not ANSI compliant and read and/or updatable 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;
```

Scroll Cursor

The SCROLL keyword creates a scroll cursor, which you can use to fetch rows of the active set in any sequence. 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. For a discussion of these abilities, see the FETCH statement on [page 2-301](#).

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. If you use a scroll cursor within a transaction, you can prevent copied rows from changing either by setting the isolation level to Repeatable Read or by locking the entire table in share mode during the transaction. (See SET ISOLATION and LOCK TABLE.)

The following example creates a scroll cursor:

```
DECLARE sc_cur SCROLL CURSOR FOR
  SELECT * FROM orders
```

Hold Cursor

If you use the WITH HOLD keywords, you create a hold cursor. A hold cursor remains open after a transaction ends. You can use the WITH HOLD keywords to declare both sequential and scroll cursors. The following example creates a hold cursor:

```
DECLARE hld_cur CURSOR WITH HOLD FOR
  SELECT customer_num, lname, city FROM customer
```

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.

You can use a 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 */
{
```

DECLARE

```
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 = '08/15/98'
        where current of c_detail;
    if(status==0) /* update was ok */
        EXEC SQL fetch c_detail into :p_orddate; /* next order */
}
if(SQLCODE==SQLNOTFOUND) /* correctly updated all found orders */
    EXEC SQL commit work; /* make updates permanent, set status */
else /* some failure in an update */
{
    save_status = SQLCODE; /* save error for loop control */
    EXEC SQL rollback work;
    SQLCODE = save_status; /* force loop to end */
}
if(SQLCODE==0) /* all updates, and the commit, worked ok */
    EXEC SQL fetch c_master into :p_custnum; /* next customer? */
}
EXEC SQL close c_master;
```

Use either the **CLOSE** statement to close the hold cursor explicitly or the **CLOSE DATABASE** or **DISCONNECT** statements to close it implicitly. The **CLOSE DATABASE** statement closes all cursors.

Declaring a Cursor as an Update or Read-Only Cursor

When you associate a cursor with a **SELECT** statement, you can define it as an update cursor or as a read-only cursor, as follows:

- Use the **FOR UPDATE** keywords to define the cursor as an update cursor.
- Use the **FOR READ ONLY** keywords to define the cursor as a read-only cursor.

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.

Defining 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 can specify particular columns that can be updated.

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.

ANSI

In an ANSI-compliant database, all simple select cursors are potentially update cursors even if they are declared without the FOR UPDATE keywords. (See the restrictions on SELECT statements associated with update cursors in [“Subset of the SELECT Statement Associated with Cursors” on page 2-252.](#)) ♦

Locking with an Update 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 you can 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.

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 statements. The columns need not be in the select list of the SELECT clause.

This column restriction applies only to UPDATE 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 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.

The following example contains ESQL/C code that uses an update cursor with a DELETE statement to delete the current row. Whenever the row is deleted, the cursor remains between rows. After you delete data, you must use a FETCH statement to advance the cursor to the next row before you can refer to the cursor in a DELETE or UPDATE statement.

```
EXEC SQL declare q_curs cursor for
      select * from customer where lname matches :last_name
      for update;

EXEC SQL open q_curs;
for (;;)
{
    EXEC SQL fetch q_curs into :cust_rec;
    if (strncmp(SQLSTATE, "00", 2) != 0)
        break;

    /* Display customer values and prompt for answer */
    printf("\n%s %s", cust_rec.fname, cust_rec.lname);
    printf("\nDelete this customer? ");
    scanf("%s", answer);

    if (answer[0] == 'y')
        EXEC SQL delete from customer where current of q_curs;
    if (strncmp(SQLSTATE, "00", 2) != 0)
        break;
}
printf("\n");
EXEC SQL close q_curs;
```

Defining a Read-Only Cursor

Use the FOR READ ONLY keywords to define a cursor as a read-only cursor. The need for the FOR READ ONLY keywords depends on whether your database is an ANSI-compliant database or a database that is not ANSI compliant.

In a database that is not ANSI compliant, the cursor that the DECLARE statement defines is a read-only cursor by default. So you do not need to specify the FOR READ ONLY keywords if you want the cursor to be a read-only cursor. The only advantage of specifying the FOR READ ONLY keywords explicitly is for better program documentation.

In an ANSI-compliant database, the cursor associated with a SELECT statement through the DECLARE statement is an update cursor by default, provided that the SELECT statement conforms to all of the restrictions for update cursors listed in [“Subset of the SELECT Statement Associated with Cursors.”](#)

Therefore, you should use the FOR READ ONLY keywords in the DECLARE statement only if you want the cursor to be a read-only cursor rather than an update cursor. You declare a read-only cursor to let the database server know that the program will not update (or delete) any row that it fetches as part of the SELECT statement. The database server can use less stringent locking for a read-only cursor than for an update cursor. ♦

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.
- In Dynamic Server with AD and XP Options, the statement cannot include the INTO EXTERNAL and INTO SCRATCH clauses. ♦

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 2-450](#).

Examples of Cursors in Non-ANSI Databases

In a database that is not ANSI compliant, a cursor associated with a SELECT statement is a read-only cursor by default. The following example declares a read-only cursor in a non-ANSI 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 shown in the following example:

```
EXEC SQL declare cust_curs cursor for
select * from customer_notansi
for read only;
```

If you want this cursor to be an update cursor, you need to specify the FOR UPDATE option in your DECLARE statement. The following example declares an update cursor:

```
EXEC SQL declare new_curs cursor for
select * from customer_notansi
for update;
```

If you want an update cursor to be able to modify only some of the columns in a table, you need to 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_notansi** table:

```
EXEC SQL declare name_curs cursor for
select * from customer_notansi
for update of fname, lname;
```

ANSI

Examples of Cursors in ANSI-compliant Databases

In an ANSI-compliant database, a cursor associated with a SELECT statement is an update cursor by default. The following example declares an update cursor in an ANSI-compliant database:

```
EXEC SQL declare x_curs cursor for
select * from customer_ansi;
```

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 shown in the following example:

```
EXEC SQL declare x_curs cursor for
      select * from customer_ansi
      for update;
```

If you want an update cursor to be able to modify only some of the columns in a table, you must specify these columns in the FOR UPDATE option. The following example declares an update cursor and specifies that this cursor can update only the **fname** and **lname** columns in the **customer_ansi** table:

```
EXEC SQL declare y_curs cursor for
      select * from customer_ansi
      for update of fname, lname;
```

If you want a cursor to be a read-only cursor, you must override the default behavior of the DECLARE statement by specifying the FOR READ ONLY option in your DECLARE statement. The following example declares a read-only cursor:

```
EXEC SQL declare z_curs cursor for
      select * from customer_ansi
      for read only;
```

Associating a Cursor with a Prepared Statement

The PREPARE statement lets you assemble the text of an SQL statement at runtime and pass the statement text to the database server for execution. If you anticipate that a dynamically prepared SELECT statement or EXECUTE PROCEDURE statement that returns values could produce more than one row of data, the prepared statement must be associated with a cursor.

The result of a PREPARE statement is a statement identifier (*statement_id* or *statement_id_var*), which is a data structure that represents the prepared statement text. To declare a cursor for the statement text, associate a cursor with the statement identifier.

You can associate a sequential cursor with any prepared SELECT or EXECUTE PROCEDURE statement.

You cannot associate a scroll cursor with a prepared INSERT statement or 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 ESQ/C code that prepares a SELECT statement and declares a 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 ESQ/C example shows:

```
EXEC SQL prepare sel_1 from 'select * from customer for update';
EXEC SQL declare sel_curs cursor for sel_1;
```

Using Cursors with Transactions

To roll back a modification, you must perform the modification within a transaction. A transaction in a database that is not ANSI-compliant begins only when the BEGIN WORK statement is executed.

ANSI

In an ANSI-compliant database, transactions are always in effect. ♦

The database server enforces the following guidelines for select 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 uses an update cursor within a transaction:

```
EXEC SQL declare q_curs cursor for
      select customer_num, fname, lname from customer
      where lname matches :last_name
      for update;
EXEC SQL open q_curs;
EXEC SQL begin work;
EXEC SQL fetch q_curs into :cust_rec; /* fetch after begin */
EXEC SQL update customer set lname = 'Smith'
      where current of q_curs;
/* no error */
EXEC SQL commit work;
```

When you update a row within a transaction, the row remains locked until the cursor is closed or the transaction is committed or rolled back. If you update a row when no transaction is in effect, the row lock is released when the modified row is written to disk.

If you update or delete a row outside a transaction, you cannot roll back the operation.

A cursor that is declared for insert is an insert cursor. In a database that uses transactions, you cannot open an insert cursor outside a transaction unless it was also declared with hold.

Subset of INSERT Associated with a Sequential Cursor

To create an insert cursor, you associate a sequential cursor with a restricted form of the INSERT statement. The INSERT statement must include a VALUES clause; it cannot contain an embedded SELECT statement.

The following example contains ESQL/C code that declares an insert cursor:

```
EXEC SQL declare ins_cur cursor for
      insert into stock values
      (:stock_no, :manu_code, :descr, :u_price, :unit, :u_desc);
```

The insert cursor simply inserts rows of data; it cannot be used to fetch data. When an insert cursor is opened, a buffer is created in memory to hold a block of rows. The buffer receives rows of data as the program executes PUT statements. The rows are written to disk only when the buffer is full. You can 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. 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.

Using an Insert Cursor with Hold

If you associate a hold cursor with an INSERT statement, you can use transactions to break a long series of PUT statements into smaller sets of PUT statements. Instead of waiting for the PUT statements to fill the buffer and 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.

References

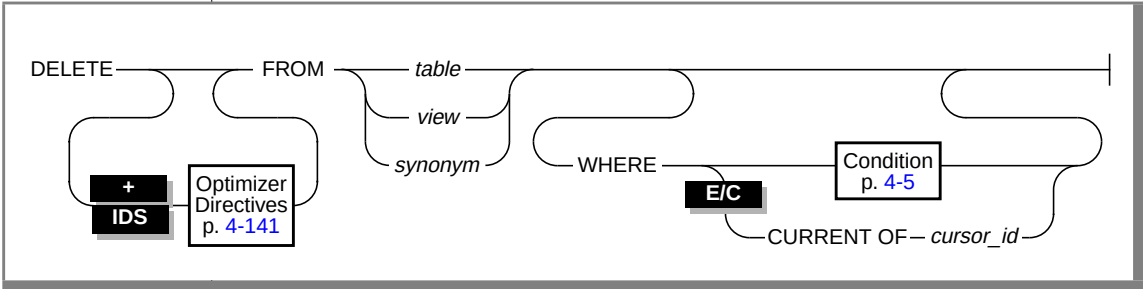
Related statements: CLOSE, DELETE, EXECUTE PROCEDURE, FETCH, FREE, INSERT, OPEN, PREPARE, PUT, SELECT, and UPDATE

For discussions of cursors and data modification, see the [Informix Guide to SQL: Tutorial](#).

DELETE

Use the DELETE statement to delete one or more rows from a table.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of the cursor whose current row is to be deleted	The cursor must have been previously declared in a DECLARE statement with a FOR UPDATE clause.	Identifier, p. 4-113
<i>synonym</i>	Name of the synonym that contains the row or rows to be deleted	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table that contains the row or rows to be deleted	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view that contains the row or rows to be deleted	The view must exist.	Database Object Name, p. 4-25

Usage

If you use the DELETE statement without a WHERE clause, all the rows in the table 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.

AD/XP

The database server locks each row affected by a DELETE statement within a transaction for the duration of the transaction. The type of lock that the database server uses is determined by the lock mode of the table, as set by a CREATE TABLE or ALTER TABLE statement, as follows:

- If the lock mode is ROW, the database server acquires one lock for each row affected by the delete.
- In Dynamic Server with AD and XP Options, if the lock mode is PAGE, the database server acquires one lock for each page affected by the delete. ♦

If the number of rows affected is very large and the lock mode is ROW, you might exceed the limits your operating system places on the maximum number of simultaneous locks. If this occurs, you can either reduce the scope of the DELETE statement or change the lock mode on the table.

If you specify a view name, the view must be updatable. For an explanation of an updatable view, see [“Updating Through Views” on page 2-235](#).

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

In an ANSI-compliant database, statements are always in an implicit transaction. Therefore, you cannot have a DELETE statement outside a transaction. ♦

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. ♦

Deleting and the WHERE Clause

If you use a WHERE clause when you delete from a table and no rows are found, no error is returned. If you delete from a table with a WHERE clause in a multistatement prepare and no rows are found, you receive a **SQLSTATE** code of 02000.

ANSI

In an ANSI-compliant database, if you use a WHERE clause when you delete from a table and no rows are found, you can detect this condition with the GET DIAGNOSTICS statement. The GET DIAGNOSTICS statement retrieves the contents of the **SQLSTATE** variable and returns the **SQLSTATE** code 02000. ♦

For additional information about **SQLSTATE** codes and a list of **SQLSTATE** codes, see the GET DIAGNOSTICS statement in this manual.

You can also use the SQLCODE field of **sqlca** to determine the same results. If you use a WHERE clause to delete rows from a table and no rows are found, the database server sets the SQLCODE field to a value of 0 (success).

ANSI

In an ANSI-compliant database, if you use a WHERE clause when you delete from a table, and no rows are found, the database server sets the SQLCODE field to a value of 100 (no matching records found). ♦

E/C

CURRENT OF Clause in ESQL/C

In ESQL/C, to use the CURRENT OF clause, you must have previously used the DECLARE statement with the FOR UPDATE clause to announce the *cursor name*.

If you use the 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.

ANSI

All select cursors are potentially update cursors in an ANSI-compliant database. You can use the CURRENT OF clause with any select cursor. ♦

References

Related Statements: INSERT, UPDATE, DECLARE, GET DIAGNOSTICS, and FETCH

For discussions of the DELETE statement, cursors, and the SQLCODE code, see the [Informix Guide to SQL: Tutorial](#).

For a discussion of the GLS aspects of the DELETE statement, see the [Informix Guide to GLS Functionality](#).

+

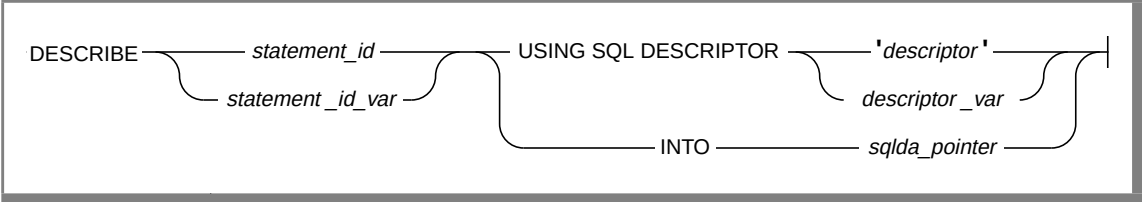
E/C

DESCRIBE

Use the DESCRIBE statement to obtain information about a prepared statement before you execute it. The DESCRIBE statement returns the prepared statement type. For an EXECUTE PROCEDURE, INSERT, SELECT, or UPDATE statement, the DESCRIBE statement also returns the number, data types and size of the values, and the name of the column or expression that the query returns.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 4-157
descriptor_var	Host variable that identifies a system-descriptor area	Variable must contain the name of an allocated system-descriptor area.	Name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>sqlda_pointer</i>	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 .
<i>statement_id</i>	Statement identifier for a prepared SQL statement	The statement identifier must be defined in a previous PREPARE statement.	PREPARE, p. 2-403
<i>statement_id_var</i>	Host variable that contains a statement identifier for a prepared SQL statement	The statement identifier must be defined in a previous PREPARE statement. The variable must be a character data type.	Name must conform to language-specific rules for variable names.

(2 of 2)

Usage

The DESCRIBE statement allows you to determine, at runtime, the type of statement that has been prepared and the number and types of data that a prepared query returns when it is executed. 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 field of the **sqlca** to indicate the statement type (that is, the keyword with which the statement begins). If the prepared statement text contains more than one SQL statement, the DESCRIBE statement returns the type of the first statement in the text.

SQLCODE is set to zero to indicate a SELECT statement *without* an INTO TEMP clause. This situation is the most common. For any other SQL statement, SQLCODE is set to a positive integer.

You can test the number against the constant names that are defined. In ESQL/C, the constant names are defined in the **sqltype.h** header file.

The DESCRIBE statement uses the SQLCODE field differently from any other statement, possibly returning a nonzero value when it executes successfully. You can revise standard error-checking routines to accommodate this behavior, if desired.

Checking for Existence of a WHERE Clause

Without a WHERE clause, the update or delete action is applied to the entire table. Check the **sqlca.sqlwarn.sqlwarn4** variable to avoid unintended global changes to your table.

If the DESCRIBE statement detects that a prepared statement contains an UPDATE or DELETE statement without a WHERE clause, the DESCRIBE statement sets the **sqlca.sqlwarn.sqlwarn4** variable to W.

Describing EXECUTE PROCEDURE, INSERT, SELECT, or UPDATE

If the prepared statement text includes a SELECT statement without an INTO TEMP clause, an EXECUTE PROCEDURE statement, an INSERT statement, or an UPDATE statement, the DESCRIBE statement also returns a description of each column or expression that is included in the EXECUTE PROCEDURE, INSERT, SELECT, or UPDATE list. These descriptions are stored in a system-descriptor area or in a pointer to an **sqlda** structure.

The description includes the following information:

- The data type of the column, as defined in the table
- The length of the column, in bytes
- The name of the column or expression

If the prepared statement is an INSERT or an UPDATE statement, the DESCRIBE statement returns only the dynamic parameters in that statement, that is, only those parameters that are expressed with a question mark (?).

You can modify the system-descriptor-area information and use it in statements that support a USING SQL DESCRIPTOR clause, such as EXECUTE, FETCH, OPEN, and PUT. 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.

Using the USING SQL DESCRIPTOR Clause

The USING SQL DESCRIPTOR clause lets you store the description of an EXECUTE PROCEDURE, INSERT, SELECT, or UPDATE list in a system-descriptor area that you allocate using an ALLOCATE DESCRIPTOR statement. You can obtain information about the resulting columns of a prepared statement through a system-descriptor area. Use the USING SQL DESCRIPTOR keywords and a descriptor to point to a system-descriptor area instead of to an **sqllda** structure.

The DESCRIBE statement sets the COUNT field in the system-descriptor area to the number of values in the EXECUTE PROCEDURE, INSERT, SELECT, or UPDATE list. If COUNT is greater than the number of item descriptors in the system-descriptor area, the system returns an error. Otherwise, the TYPE, LENGTH, NAME, SCALE, PRECISION, and NULLABLE information is set and memory for DATA fields is allocated automatically.

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.

The following examples show the use of a system descriptor in a DESCRIBE statement. In the first example, the descriptor is a quoted string; in the second example, it is an embedded variable name.

```
main()
{
    . . .
    EXEC SQL allocate descriptor 'desc1' with max 3;
    EXEC SQL prepare curs1 FROM 'select * from tab';
    EXEC SQL describe curs1 using sql descriptor 'desc1';
}

EXEC SQL describe curs1 using sql descriptor :desc1var;
```


Using the INTO `sqlda` pointer Clause

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. The DESCRIBE statement sets the **sqlda.sqlld** field to the number of values in the EXECUTE PROCEDURE, INSERT, SELECT, or UPDATE list. The **sqlda** structure also contains an array of data descriptors (**sqlvar** structures), one for each value in the EXECUTE PROCEDURE, INSERT, SELECT, or UPDATE list. After a DESCRIBE statement is executed, the **sqlda.sqlvar** structure has the **sqltype**, **sqllen**, and **sqlname** fields set.

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.

References

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

For a task-oriented discussion of the DESCRIBE statement, see the [Informix Guide to SQL: Tutorial](#).

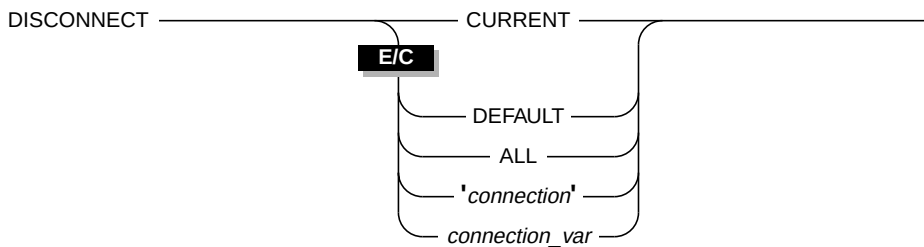
For more information about how to use a system-descriptor area and **sqlda**, refer to the [INFORMIX-ESQL/C Programmer's Manual](#).

+

DISCONNECT

Use the DISCONNECT statement to terminate a connection between an application and a database server.

Syntax



Element	Purpose	Restrictions	Syntax
<i>connection</i>	String that identifies a connection to be terminated	The specified connection must match a connection name assigned by the CONNECT statement.	Quoted String, p. 4-157
<i>connection_var</i>	Host variable that holds the value of <i>connection</i>	Variable must be a fixed-length character data type. The specified connection 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.

In ESQL/C, if you disconnect a specific connection with *connection* or *connection_var*, 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 with the DATABASE or CREATE DATABASE statements)

You can use DISCONNECT to disconnect the default connection. For more information, see [“DEFAULT Option” on page 2-79](#) and [“The Implicit Connection with DATABASE Statements” on page 2-80](#).

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 2-79](#) and [“The Implicit Connection with DATABASE Statements” on page 2-80](#) 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.

E/C

Disconnecting in a Thread-Safe Environment

If you issue the **DISCONNECT** statement in a thread-safe ESQL/C application, keep in mind that an active connection can only be disconnected from within the thread in which it is active. Therefore one thread cannot disconnect another thread's active connection. 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 2-505](#) 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
```

In ESQL/C, the ALL keyword has the same effect on multithreaded applications that it has on single-threaded applications. Execution of the DISCONNECT ALL statement causes all connections in all threads to be terminated. However, the DISCONNECT ALL statement fails if any of the connections is in use or has an ongoing transaction associated with it. If either of these conditions is true, none of the connections is disconnected. ♦

References

Related statements: CONNECT, DATABASE, and SET CONNECTION

For information on multithreaded applications, see the [INFORMIX-ESQL/C Programmer's Manual](#).

+

DROP DATABASE

Use the DROP DATABASE statement to delete an entire database, including all system catalog tables, indexes, and data.

Syntax

DROP DATABASE

Database Name p. 4-22

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.

During a DROP DATABASE operation, the database server acquires a lock on each table in the database and holds the locks until the entire operation is complete. Configure your database server with enough locks to accommodate this fact. For example, if the database to be dropped has 2500 tables, but less than 2500 locks have been configured for your server, your DROP DATABASE statement will fail. For further information on how to set the number of locks available to the database server, see your [Administrator's Guide](#).

The following statement drops the **stores7** database:

```
DROP DATABASE stores7
```

In DB-Access, use this statement with caution. DB-Access does not prompt you to verify that you want to delete the entire database. ♦

References

Related statements: CREATE DATABASE and CONNECT

+

DROP INDEX

Use the DROP INDEX statement to remove an index.

Syntax

```
DROP INDEX _____ index _____|
```

Element	Purpose	Restrictions	Syntax
<i>index</i>	Name of the index to drop	The index must exist.	Database Object Name, p. 4-25

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 this: “121_13” (quotes used to show the spaces).

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

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

For information on the performance characteristics of indexes, see your [*Performance Guide*](#).

+

DROP PROCEDURE

Use the DROP PROCEDURE statement to remove a stored procedure from the database.

Syntax

```
DROP PROCEDURE _____ procedure _____ |
```

Element	Purpose	Restrictions	Syntax
<i>procedure</i>	Name of the procedure to drop	The procedure must exist in the current database.	Identifier, p. 4-113

Usage

You must be the owner of the stored procedure or have the DBA privilege to use the DROP PROCEDURE statement.

Dropping the stored procedure removes the text and executable versions of the procedure.

Tip: *You cannot drop a stored procedure from within the same procedure.*

References

Related statement: CREATE PROCEDURE

For a discussion of how to use stored procedures, see the [Informix Guide to SQL: Tutorial](#).



+

IDS

DROP ROLE

Use the DROP ROLE statement to remove a previously created role.
You can use this statement only with Dynamic Server.

Syntax

DROP ROLE _____ *role* _____

Element	Purpose	Restrictions	Syntax
<i>role</i>	Name of the role to drop	The role name must have been created with the CREATE ROLE statement.	Identifier, p. 4-113

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.

References

Related statements: CREATE ROLE, GRANT, REVOKE, and SET ROLE

For a discussion of how to use roles, see the [Informix Guide to SQL: Tutorial](#).

+

DROP SYNONYM

Use the DROP SYNONYM statement to remove a previously defined synonym.

Syntax

```
DROP SYNONYM _____ synonym _____
```

Element	Purpose	Restrictions	Syntax
<i>synonym</i>	Name of the synonym to drop	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25

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 database object the name of the dropped table. The old synonym then refers to the new database object. For a complete discussion of synonym chaining, see the CREATE SYNONYM statement.

References

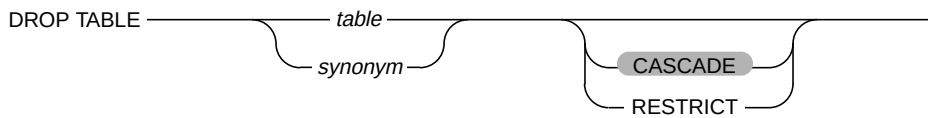
Related statement: CREATE SYNONYM

For a discussion of synonyms, see the [Informix Guide to SQL: Tutorial](#).

DROP TABLE

Use the DROP TABLE statement to remove a table, along with its associated indexes and data.

Syntax



Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of the table to drop	The table must exist.	Database Object Name, p. 4-25
<i>synonym</i>	Name of the synonym to drop	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25

Usage

You must be the owner of the table or have the DBA privilege to use the DROP TABLE statement.

If you are using Dynamic Server with AD and XP Options, you cannot drop a table that includes a dependent GK index unless the dependent index is entirely dependent on the affected table. ♦

If you issue a DROP TABLE statement, DB-Access does not prompt you to verify that you want to delete an entire table. ♦

Effects of 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.

When you drop a table, you do not remove any synonyms for the table that have been created in an external database. If you want to remove external synonyms to the dropped table, you must do so manually with the DROP SYNONYM statement.

Specifying CASCADE Mode

The CASCADE mode means that a DROP TABLE statement removes related database objects, including referential constraints built on the table, views defined on the table, and any violations and diagnostics tables associated with the table. 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 tables that have referential constraints and views defined on the table or 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 existing referential constraints reference *table name* or if any existing views are defined on *table name* or if any violations and diagnostics tables are associated with *table name*.

Tables That Cannot Be Dropped

Observe the following restrictions on the types of tables that you can drop:

- You cannot drop any system catalog tables.
- You cannot drop a table that is not in the current database.
- You cannot drop a violations 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.
- If you are using Dynamic Server with AD and XP Options, you cannot drop a table that appears in the FROM clause of a GK index. ♦

Examples of Dropping a Table

The following example deletes two tables. Both tables are within the current database and are owned by the current user. Neither table has a violations or diagnostics table associated with it. Neither table has a referential constraint or view defined on it.

```
DROP TABLE customer;  
DROP TABLE stores7@acctgtg:joed.state;
```

References

Related statements: CREATE TABLE and DROP DATABASE

For a discussion of the data integrity of tables, see the [Informix Guide to SQL: Tutorial](#).

For a discussion of how to create a table, see the [Informix Guide to Database Design and Implementation](#).

+

IDS

DROP TRIGGER

Use the DROP TRIGGER statement to remove a trigger definition from a database.

You can this statement only with Dynamic Server.

Syntax

DROP TRIGGER _____ *trigger* _____

Element	Purpose	Restrictions	Syntax
<i>trigger</i>	Name of the trigger to drop	The trigger must exist.	Identifier, p. 4-113

Usage

You must be the owner of the trigger or have the DBA privilege to drop the trigger.

Dropping a trigger removes the text of the trigger definition and the executable trigger from the database.

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

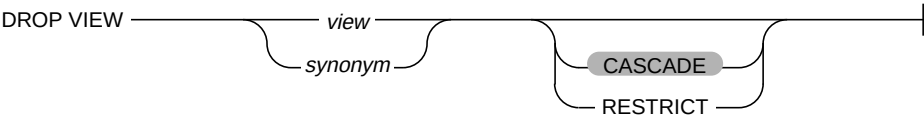
Related statements: CREATE PROCEDURE and CREATE TRIGGER

+

DROP VIEW

Use the DROP VIEW statement to remove a view from a database.

Syntax



Element	Purpose	Restrictions	Syntax
<i>synonym</i>	Name of the synonym to drop	The synonym and the view to which the synonym points must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view to drop	The view must exist.	Database Object Name, p. 4-25

Usage

You must be the owner of the trigger or have the DBA privilege to drop the trigger.

When you drop a view or its synonym, 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.

References

Related statements: DROP VIEW and DROP TABLE

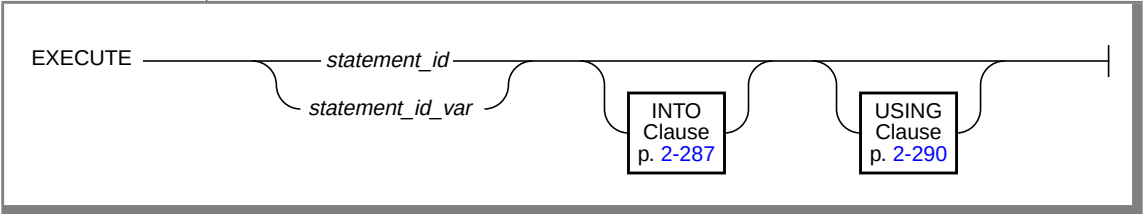
For a discussion of views, see the [*Informix Guide to SQL: Tutorial*](#).

EXECUTE

Use the EXECUTE statement to run a previously prepared statement or set of statements.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>statement_id</i>	Statement identifier for a prepared 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. 2-403
<i>statement_id_var</i>	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. 2-403

Usage

The EXECUTE statement passes a prepared SQL statement to the database server for execution. If the statement contained question mark (?) placeholders, specific values are supplied for them before execution. Once prepared, an SQL statement can be executed as often as needed.

You can execute any prepared statement. However, for stored procedures that return more than one row, you cannot execute a prepared `SELECT` statement or a prepared `EXECUTE PROCEDURE` statement. 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. If you prepare an `EXECUTE PROCEDURE` statement for a procedure that returns multiple rows, you need to use the `DECLARE`, `OPEN`, and `FETCH` cursor statements just as you would with a `SELECT` statement.

If you create or drop a trigger after you prepare a triggering `INSERT`, `DELETE`, or `UPDATE` statement, the prepared statement returns an error when you execute it.

The following example shows an `EXECUTE` statement within an `ESQL/C` program:

```
EXEC SQL prepare del_1 from
      'delete from customer
        where customer_num = 119';
EXEC SQL execute del_1;
```

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.

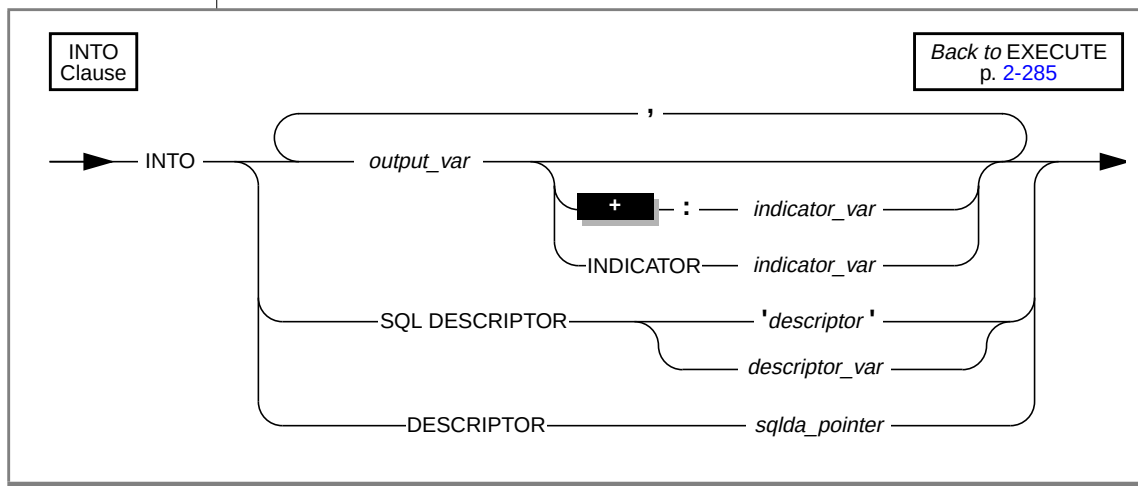
The *sqlca* Record and EXECUTE

Following an EXECUTE statement, the **sqlca** can reflect two results:

- The **sqlca** can reflect an error within the EXECUTE statement. For example, when an UPDATE ...WHERE... statement within a prepared statement processes zero rows, the database server sets **sqlca** to 100.
- The **sqlca** can also reflect the success or failure of the executed statement.

INTO Clause

The INTO clause allows you to execute a prepared singleton SELECT statement or a prepared EXECUTE PROCEDURE statement, and store the returned values into output variables, output SQL descriptors, or output **sqlca** pointers.



Element	Purpose	Restrictions	Syntax
<i>descriptor</i>	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 4-157
<i>descriptor_var</i>	Host variable that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 4-157
<i>indicator_var</i>	Host variable that receives a return code if null data is placed in the corresponding <i>output_var</i>	Variable cannot be DATETIME or INTERVAL data type.	Variable name must conform to language-specific rules for variable names.
<i>output_var</i>	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.
<i>sqlda_pointer</i>	Pointer 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 <i>sqlda_pointer</i> with a dollar sign (\$) or a colon (:). You must use an sqlda structure if you are using dynamic SQL statements.	DESCRIBE, p. 2-262

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 from a table.

Restrictions with the INTO Clause

If you execute a prepared SELECT statement that returns more than one row of data, you receive an error message. In addition, if you prepare and declare a statement, and then attempt to execute that statement, you receive an error message.

You cannot select a null value from a table column and place that value into an output variable. If you know in advance that a table column contains a null value, after you select the data, check the indicator variable that is associated with the column to determine if the value is null.

The following list describes the procedure for how to use the INTO clause with the EXECUTE statement:

1. Declare the output variables that the EXECUTE statement uses.
2. Use the PREPARE statement to prepare your SELECT statement or to prepare your EXECUTE PROCEDURE statement.
3. Use the EXECUTE statement, with the INTO clause, to execute your SELECT statement or to execute your EXECUTE PROCEDURE statement.

The following example shows how to use the INTO clause with an EXECUTE statement in ESQL/C:

```
EXEC SQL prepare sell from 'select fname, lname from customer
    where customer_num =123';
EXEC SQL execute sell into :fname, :lname using :cust_num;
```

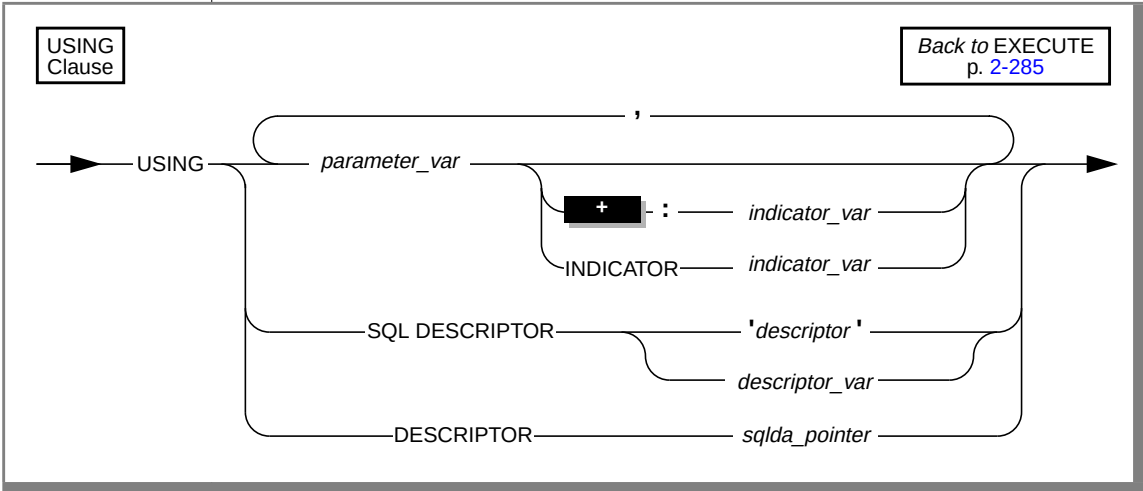
The following example shows how to use the INTO clause to return multiple rows of data:

```
EXEC SQL BEGIN DECLARE SECTION;
int customer_num =100;
char fname[25];
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare sell from 'select fname from customer
    where customer_num=?';
for ( ;customer_num < 200; customer_num++)
{
    EXEC SQL execute sell into :fname using customer_num;
    printf("Customer number is %d\n", customer_num);
    printf("Customer first name is %s\n\n", fname);
}
```


USING Clause

The USING clause specifies values to replace question-mark (?) placeholders in the prepared statement. Providing values in the EXECUTE statement that replace the question-mark placeholders in the prepared statement is sometimes called *parameterizing* the prepared statement.



Element	Purpose	Restrictions	Syntax
<i>descriptor</i>	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated. Make sure surrounding quotes are single.	Quoted String, p. 4-157
<i>descriptor_var</i>	Host variable that identifies a system-descriptor area	System-descriptor area must already be allocated.	Name must conform to language-specific rules for variable names.

Element	Purpose	Restrictions	Syntax
<i>indicator_var</i>	Host variable that receives a return code if null data is placed in the corresponding <i>parameter_var</i> This variable receives truncation information if truncation occurs.	Variable cannot be DATETIME or INTERVAL data type.	Name must conform to language-specific rules for variable names.
<i>parameter_var</i>	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Name must conform to language-specific rules for variable names.
<i>sqlda_pointer</i>	Pointer 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 <i>sqlda_pointer</i> with a dollar sign (\$) or a colon (:). You must use an sqlda structure if you are using dynamic SQL statements.	DESCRIBE, p. 2-262

(2 of 2)

You can specify any of the following items to replace the question-mark placeholders in a statement before you execute it:

- A host variable name (if the number and data type of the question marks are known at compile time)
- A system descriptor that identifies a system
- A descriptor that is a pointer to an **sqlda** structure

Supplying Parameters Through Host or Program Variables

You must supply one parameter 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 ESQL/C:

```
stcopy ("update orders set order_date = ? where po_num = ?", stmt1);
EXEC SQL prepare statement_1 from :stmt1;
EXEC SQL execute statement_1 using :order_date :po_num;
```

Supplying Parameters Through a System Descriptor

You can create a system-descriptor area that describes the data type and memory location of one or more values and then specify the descriptor in the USING SQL DESCRIPTOR clause of the EXECUTE statement.

Each time that the EXECUTE statement is run, the values that the system-descriptor area describes are used to replace question-mark (?) placeholders in the PREPARE statement.

The COUNT field corresponds to the number of dynamic parameters in the prepared statement. The value of COUNT must be less than or equal to the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.

The following example shows how to use system descriptors to execute a prepared statement in ESQL/C:

```
EXEC SQL execute prep_stmt using sql descriptor 'desc1';
```

Supplying Parameters Through an sqlda Structure

You can specify the **sqlda** pointer in the USING DESCRIPTOR clause of the EXECUTE statement. Each time the EXECUTE statement is run, the values that the **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 ESQL/C:

```
EXEC SQL execute prep_stmt using descriptor pointer2
```

Error Conditions with EXECUTE

If a prepared statement fails to access any rows, the database server returns (0). In a multistatement prepare, if any statement in the following list fails to access rows, the database server returns SQLNOTFOUND (100):

- INSERT INTO *table* SELECT...WHERE...
- SELECT INTO TEMP...WHERE...
- DELETE...WHERE
- UPDATE...WHERE...

ANSI

In an ANSI-compliant database, if you prepare and execute any of the statements in the preceding list, and no rows are returned, the database server returns SQLNOTFOUND (100). ♦

References

Related statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, EXECUTE IMMEDIATE, FETCH, PREPARE, PUT, and SET DESCRIPTOR

For a task-oriented discussion of the EXECUTE statement, see the [Informix Guide to SQL: Tutorial](#).

For more information about concepts relating to the EXECUTE statement, refer to the [INFORMIX-ESQL/C Programmer's Manual](#).

+

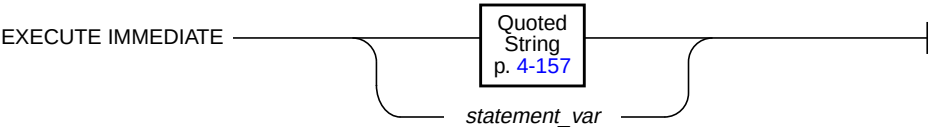
E/C

EXECUTE IMMEDIATE

Use the EXECUTE IMMEDIATE statement to perform the functions of the PREPARE, EXECUTE, and FREE statements.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
statement_var	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 2-295 and “Restrictions on Allowed Statements” on page 2-296.	Name must conform to language-specific rules for variable names.

Usage

The EXECUTE IMMEDIATE statement makes it easy to execute dynamically a single simple SQL statement that is constructed during program execution. For example, you can obtain the name of a database from program input, construct the DATABASE statement as a program variable, and then use EXECUTE IMMEDIATE to execute the statement, which opens the database.

The quoted string is a character string that includes one or more SQL statements. The string, or the contents of *statement_var*, is parsed and executed if correct; then all data structures and memory resources are released immediately. In the usual method of dynamic execution, these functions are distributed among the PREPARE, EXECUTE, and FREE statements.

EXECUTE IMMEDIATE and Restricted Statements

You cannot use the EXECUTE IMMEDIATE statement to execute the following SQL statements. Although the EXECUTE PROCEDURE statement appears on this list, the restriction applies only to EXECUTE PROCEDURE statements that return values.

CLOSE	OUTPUT
CONNECT	PREPARE
DECLARE	SELECT
DISCONNECT	SET AUTOFREE
EXECUTE	SET CONNECTION
EXECUTE PROCEDURE	SET DEFERRED_PREPARE
FETCH	SET DESCRIPTOR
GET DESCRIPTOR	WHENEVER
GET DIAGNOSTICS	
OPEN	

In addition, you cannot use the EXECUTE IMMEDIATE statement to execute the following statements in text that contains multiple statements that are separated by semicolons.

CLOSE DATABASE	DROP DATABASE
CREATE DATABASE	SELECT (except SELECT INTO TEMP)
DATABASE	

Use a PREPARE statement to execute a dynamically constructed SELECT statement.

Restrictions on Allowed Statements

The following restrictions apply to the statement that is contained in the quoted string or in the statement variable:

- The statement cannot contain a host-language comment.
- Names of host-language variables are not recognized as such in prepared text. The only identifiers that you can use are names defined in the database, such as table names and columns.
- The statement cannot reference a host variable list or a descriptor; it must not contain any question-mark (?) placeholders, which are allowed with a PREPARE statement.
- The text must not include any embedded SQL statement prefix, such as the dollar sign (\$) or the keywords EXEC SQL. Although it is not required, the SQL statement terminator (;) can be included in the statement text.

Examples of the EXECUTE IMMEDIATE Statement

The following examples show EXECUTE IMMEDIATE statements in ESQL/C. Both examples use host variables that contain a CREATE DATABASE statement. The first example uses the SQL statement terminator (;) inside the quoted string.

```
sprintf(cdb_text1, "create database %s;", usr_db_id);  
EXEC SQL execute immediate :cdb_text;
```

```
sprintf(cdb_text2, "create database %s", usr_db_id);  
EXEC SQL execute immediate :cdb_text;
```

References

Related statements: EXECUTE, FREE, and PREPARE

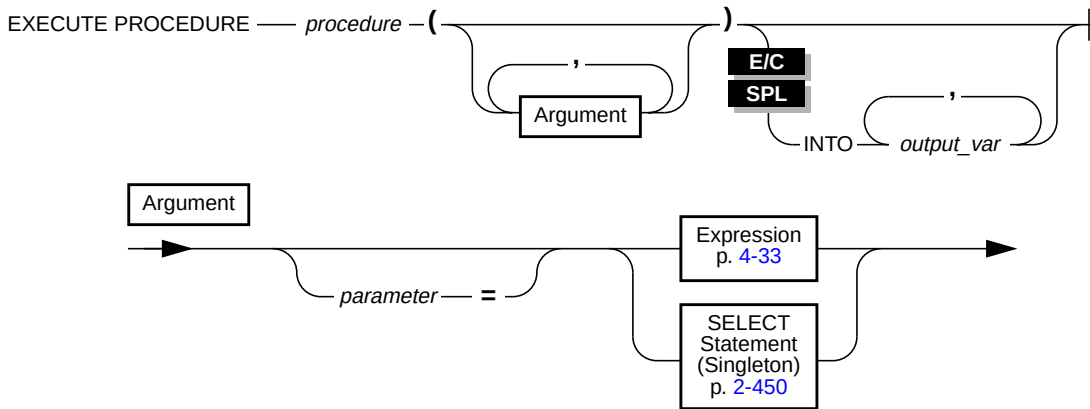
For a discussion of quick execution, see the [Informix Guide to SQL: Tutorial](#).

+

EXECUTE PROCEDURE

Use the EXECUTE PROCEDURE statement to execute a procedure from the DB-Access interactive editor, ESQL / C, or another stored procedure.

Syntax



Element	Purpose	Restrictions	Syntax
<i>output_var</i>	Host variable that receives a returned value from a procedure	If you issue an EXECUTE PROCEDURE statement in ESQL / C, the receiving variables must be host variables. If you issue an EXECUTE PROCEDURE statement in a stored procedure, the receiving variables must be procedure variables. If you issue an EXECUTE PROCEDURE statement in a CREATE TRIGGER statement, the receiving variables must be column names in the triggering table or another table.	Name must conform to language-specific rules for variable names. Expression, p. 4-33 Identifier, p. 4-113
<i>parameter</i>	Name of a parameter for which you supply an argument to the procedure	The parameter name must match the parameter name that you specified in a corresponding CREATE PROCEDURE statement. If you use the <i>parameter</i> = syntax for any argument in the EXECUTE PROCEDURE statement, you must use it for all arguments.	Expression, p. 4-33

Usage

The EXECUTE PROCEDURE statement invokes the procedure called *procedure*.

If an EXECUTE PROCEDURE statement has more arguments than the called procedure expects, an error is returned.

If an EXECUTE PROCEDURE statement has fewer arguments than the called procedure expects, the arguments are said to be missing. Missing arguments are initialized to their corresponding default values if default values were specified. (See the CREATE PROCEDURE statement on [page 2-134](#).) This initialization occurs before the first executable statement in the body of the procedure.

If arguments are missing and do not have default values, they are initialized to the value of UNDEFINED. An attempt to use any variable that has the value of UNDEFINED results in an error.

Name or position, but not both, binds procedure arguments to procedure parameters. That is, you can use *parameter* = syntax for none or all of the arguments that are specified in one EXECUTE PROCEDURE statement.

For instance, in the following example, both the procedure calls are valid for a procedure that expects three character arguments, t, d, and n:

```
EXECUTE PROCEDURE add_col (t = 'customer', d = 'integer', n = 'newint')
EXECUTE PROCEDURE add_col ('customer', 'newint', 'integer')
```

E/C

In ESQL/C, if the EXECUTE PROCEDURE statement returns more than one row, it must be enclosed within an SPL FOREACH loop or accessed through a cursor. ♦

E/C

INTO Clause

SPL

In ESQL/C programs and stored procedures, the INTO clause specifies where the values that the procedure returns are stored.

The *output_var* list is a list of the host variables that receive the returned values from a procedure call. A procedure that returns more than one row must be enclosed in a cursor.

If you execute a procedure from within a stored procedure, the list contains procedure variables.

If you execute a procedure from within a triggered action, the list contains column names from the triggering table or another table. For information on triggered actions, see the CREATE TRIGGER statement on [page 2-198](#).

You cannot prepare an EXECUTE PROCEDURE statement that has an INTO clause. For more information, see [“Executing Stored Procedures Within a PREPARE Statement” on page 2-407](#).

References

Related statements: CREATE PROCEDURE, DROP PROCEDURE, GRANT, and CALL

For a discussion of how to create and use stored procedures, see the [Informix Guide to SQL: Tutorial](#).

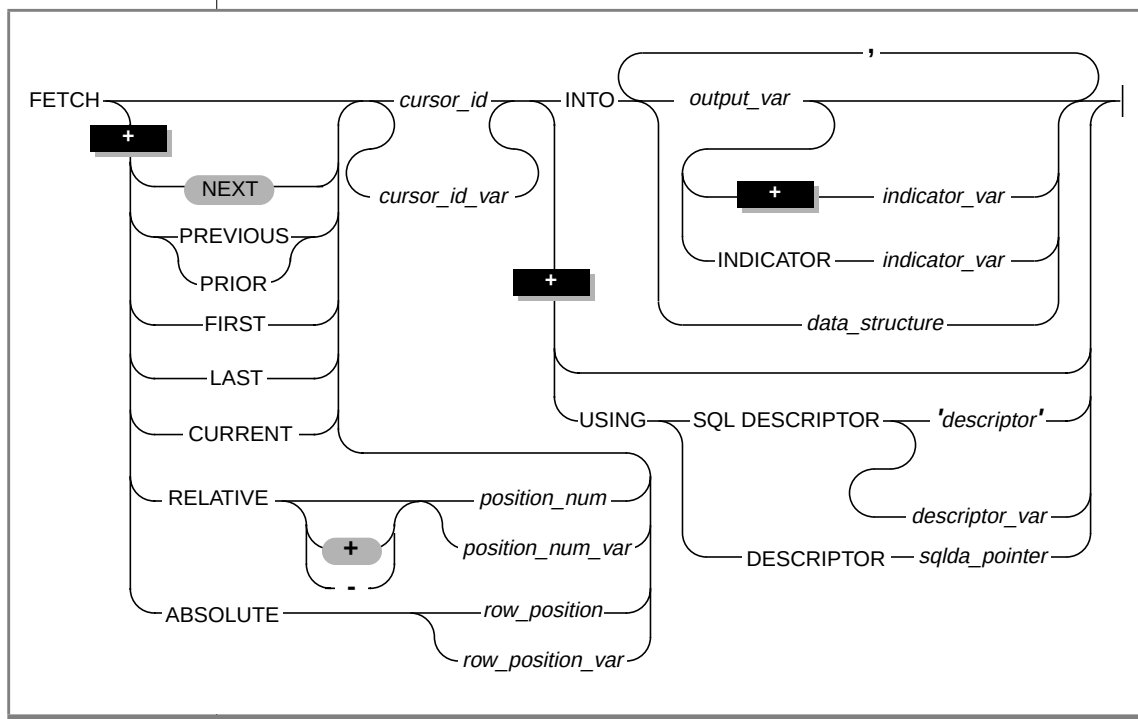
E/C

FETCH

Use the FETCH statement to move a cursor to a new row in the active set and to retrieve the row values from memory.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of a cursor from which rows are to be retrieved	The cursor must have been created in an earlier DECLARE statement and opened in an earlier OPEN statement.	Identifier, p. 4-113
<i>cursor_id_var</i>	Host variable name that holds the value of <i>cursor_id</i>	The identified cursor must have been created in an earlier DECLARE statement and opened in an earlier OPEN statement.	Name must conform to language-specific rules for variable names.
<i>data_structure</i>	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 <i>data_structure</i> .	Name must conform to language-specific rules for data-structure names.
<i>descriptor</i>	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. 4-157
<i>descriptor_var</i>	Host variable name that holds the value of <i>descriptor</i>	The identified system-descriptor area must have been allocated with the ALLOCATE DESCRIPTOR statement.	Name must conform to language-specific rules for variable names.
<i>indicator_var</i>	Host variable that receives a return code if null data is placed in the corresponding <i>output_var</i>	This parameter is optional, but use an indicator variable if the possibility exists that the value of <i>output_var</i> is null. If you specify the indicator variable without the INDICATOR keyword, you cannot put a space between <i>output_var</i> and <i>indicator_var</i> . For information about ESQL / C rules for placing a prefix before the <i>indicator_var</i> , see the INFORMIX-ESQL/C Programmer's Manual .	Name must conform to language-specific rules for variable names.

Element	Purpose	Restrictions	Syntax
<i>output_var</i>	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.	Name must conform to language-specific rules for variable names.
<i>position_num</i>	Integer value that gives the relative position of the desired row in relation to the current row in the active set of rows	A value of 0 fetches the current row.	Literal Number, p. 4-139.
<i>position_num_var</i>	Host variable that contains <i>position_num</i>	A value of 0 fetches the current row.	Name must conform to language-specific rules for variable names.
<i>row_position</i>	Integer value that gives the position of the desired row in the active set of rows	The value of <i>row_position</i> must be 1 or higher.	Literal Number, p. 4-139.
<i>row_position_var</i>	Host variable that contains <i>row_position</i>	The value of <i>row_position</i> must be 1 or higher.	Name must conform to language-specific rules for variable names.
<i>sqlda_pointer</i>	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 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.)

In X/Open mode, if a cursor-direction value (such as NEXT or RELATIVE) is specified, a warning message is issued, indicating that the statement does not conform to X/Open standards. ♦

FETCH with a Sequential Cursor

A sequential cursor can fetch only the next row in sequence from the active set. The sole keyword 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 ESQL/C illustrates the use of 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 cursor can fetch any row in the active set, either by specifying an absolute row position or a relative offset. Use the following keywords 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	Retrieves the previous row in the active set. (Synonymous with PREVIOUS.)
FIRST	Retrieves the first row in the active set.
LAST	Retrieves the last row in the active set.

(1 of 2)

Keyword	Effect
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 <i>n</i> th row, relative to the current cursor position in the active set, where <i>position_num</i> (or <i>position_num_var</i>) supplies <i>n</i> . A negative value indicates the <i>n</i> th row prior to the current cursor position. If <i>position_num</i> is 0, the current row is fetched.
ABSOLUTE	Retrieves the <i>n</i> th row in the active set, where <i>row_position</i> (or <i>row_position_var</i>) supplies <i>n</i> . Absolute row positions are numbered from 1.

(2 of 2)

The following ESQL/C examples illustrate the FETCH statement:

```
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;
```

ABSOLUTE Option

The row-position values that are used in the ABSOLUTE option are valid only while the cursor is open. Do not confuse row-position values with rowid values. A rowid value is based on the position of a row in its table and remains valid until the table is rebuilt. A row-position value 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 the executed procedure must be returned into a memory location. You can specify these destinations in one of the following ways:

- Use the INTO clause of a SELECT statement.
- Use the INTO clause of a EXECUTE PROCEDURE statement.
- Use the INTO clause of a FETCH statement.
- Use a system-descriptor area.
- Use an **sqllda** structure.

Using the INTO Clause of SELECT or EXECUTE PROCEDURE

The SELECT or EXECUTE PROCEDURE statement that is associated with a cursor can contain an INTO clause, which specifies the program variables that are to receive the values. You can use this method only when the SELECT or EXECUTE PROCEDURE statement is written as part of the cursor declaration. In this case, the FETCH statement can contain an INTO clause.

The following example uses the INTO clause of the SELECT statement to specify program variables in ESQL/C:

```
EXEC SQL declare ord_date cursor for
      select order_num, order_date, po_num
      into :o_num, :o_date, :o_po;
EXEC SQL open ord_date;
EXEC SQL fetch next ord_date;
```

Use an indicator variable if the data that is returned from the SELECT statement might be null.

Using the INTO Clause of FETCH

When the SELECT statement omits the INTO clause, you must specify the destination of the data whenever a row is fetched. The FETCH statement can 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 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

You can use a system-descriptor area as an output variable. The keywords USING SQL DESCRIPTOR introduce the name of the system-descriptor area into which you fetch the contents of a row. 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 a valid FETCH USING SQL DESCRIPTOR statement:

```
EXEC SQL fetch selcurs using sql descriptor 'desc';
```

Using an sqlda Structure

You can use a pointer to an **sqlda** structure to supply destinations. This structure contains data descriptors that specify the data type and memory location for one selected value. The keywords USING DESCRIPTOR introduce the name of the **sqlda** pointer structure.

When you create a SELECT statement dynamically, you cannot use an INTO clause because you cannot name host variables in a prepared statement. If you are certain of the number and data type of values in the select list, you can use an INTO clause in the FETCH statement. However, if user input generated the query, you might not be certain of the number and data type of values that are being selected. In this case, you must use an **sqlda** pointer structure, as the following list describes:

- Use the DESCRIBE statement to fill in the **sqlda** structure.
- Allocate memory to hold the data values.
- Name the **sqlda** structure in the FETCH statement.

The following example shows a valid FETCH USING DESCRIPTOR statement:

```
EXEC SQL fetch selcurs using descriptor pointer2;
```

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.

- 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.



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.

Checking the Result of FETCH

You can use the GET DIAGNOSTICS statement to check the result of each FETCH statement. Examine the **RETURNED_SQLSTATE** field of the GET DIAGNOSTICS statement to check if the field contains the value 02000.

If a row is returned successfully, the **RETURNED_SQLSTATE** field of GET DIAGNOSTICS contains the value 00000. If no row is found, the preprocessor sets the **SQLSTATE** code to 02000, which indicates no data found, and the current row is unchanged. Five conditions set the **SQLSTATE** code to 02000, indicating no data found, as the following list describes:

- 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.

You can also use SQLCODE of **sqlca** to determine the same results.

References

Related statements: ALLOCATE DESCRIPTOR, CLOSE, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, GET DESCRIPTOR, OPEN, PREPARE, SET DEFERRED_PREPARE, and SET DESCRIPTOR

For a task-oriented discussion of the FETCH statement, see the [Informix Guide to SQL: Tutorial](#).

For more information about concepts relating to the FETCH statement, see the [INFORMIX-ESQL/C Programmer's Manual](#).

+

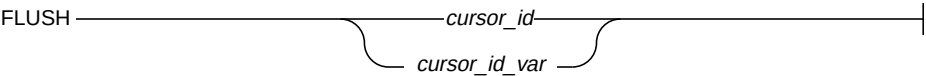
E/C

FLUSH

Use the FLUSH statement to force rows that a PUT statement buffered to be written to the database.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of a cursor	A DECLARE statement must have previously created the cursor.	Identifier, p. 4-113
<i>cursor_id_var</i>	Host variable that holds the value of <i>cursor_id</i>	Host variable must be a character data type. A DECLARE statement must have previously created the cursor.	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 automatically.

The following example shows a FLUSH statement:

```
FLUSH icurs
```

Error Checking FLUSH Statements

The **sqlca** structure contains information on the success of each FLUSH statement and the number of rows that are inserted successfully. The result of each FLUSH statement is contained in the fields of the **sqlca**: **sqlca.sqlcode**, **SQLCODE** and **sqlca.sqlerrd[2]**.

When you use data buffering with an insert cursor, you do not discover errors until the buffer is flushed. For example, an input value that is incompatible with the data type of the column for which it is intended is discovered only when the buffer is flushed. When an error is discovered, rows in the buffer that are located after the error are *not* inserted; they are lost from memory.

The **SQLCODE** field is set either to an error code or to zero 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** contains the number of rows that were successfully inserted. (Uninserted rows are discarded from the buffer.)



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

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, for example, **total** and **pending**.
2. When the cursor opens, set both variables to 0.
3. Each time a PUT statement executes, increment both **total** and **pending**.
4. Whenever a FLUSH statement executes or the cursor is closed, subtract the third field of the SQLERRD array from **pending**.

References

Related statements: CLOSE, DECLARE, OPEN, and PREPARE

For a task-oriented discussion of FLUSH, see the [*Informix Guide to SQL: Tutorial*](#).

For information about the **sqlca** structure, see the [*INFORMIX-ESQL/C Programmer's Manual*](#).

+

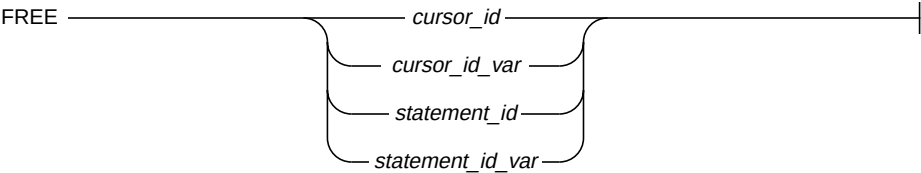
E/C

FREE

Use the FREE statement to release resources that are allocated to a prepared statement or to a cursor.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of a cursor	A DECLARE statement must have previously created the cursor.	Identifier, p. 4-113
<i>cursor_id_var</i>	Host variable that holds the value of <i>cursor_id</i>	Variable must be a character data type. Cursor must have been previously created by a DECLARE statement.	Name must conform to language-specific rules for variable names
<i>statement_id</i>	String that identifies 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. 2-403
<i>statement_id_var</i>	Host variable that identifies an SQL statement	The statement identifier must be defined in a previous PREPARE statement. The host variable must be a character data type.	PREPARE, p. 2-403

Usage

The FREE statement releases the resources that the database server and application-development tool allocated for a prepared statement or a declared cursor.

Freeing a Statement

If you prepared a statement (but did not declare a cursor for it), FREE *statement_id* (or *statement_id_var*) 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_var*) releases only the resources in the application development tool; the cursor can still be used. The resources in the database server are released only when you free the cursor.

After you free a statement, you cannot execute it or declare a cursor for it until you prepare it again.

The following ESQL/C example shows the sequence of statements that is used to free an implicitly prepared statement:

```
EXEC SQL prepare sel_stmt from 'select * from orders';
.
.
.
EXEC SQL free sel_stmt;
```

The following ESQL/C example shows the sequence of statements that are used to release the resources of an explicitly prepared statement. The first FREE statement in this example frees the cursor. The second FREE statement in this example frees the prepared statement.

```
printf(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_var`).

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 2-316](#).

References

Related statements: `CLOSE`, `DECLARE`, `EXECUTE`, `EXECUTE IMMEDIATE`, `PREPARE`, and `SET AUTOFREE`

For a task-oriented discussion of the `FREE` statement, see the [Informix Guide to SQL: Tutorial](#).

+

E/C

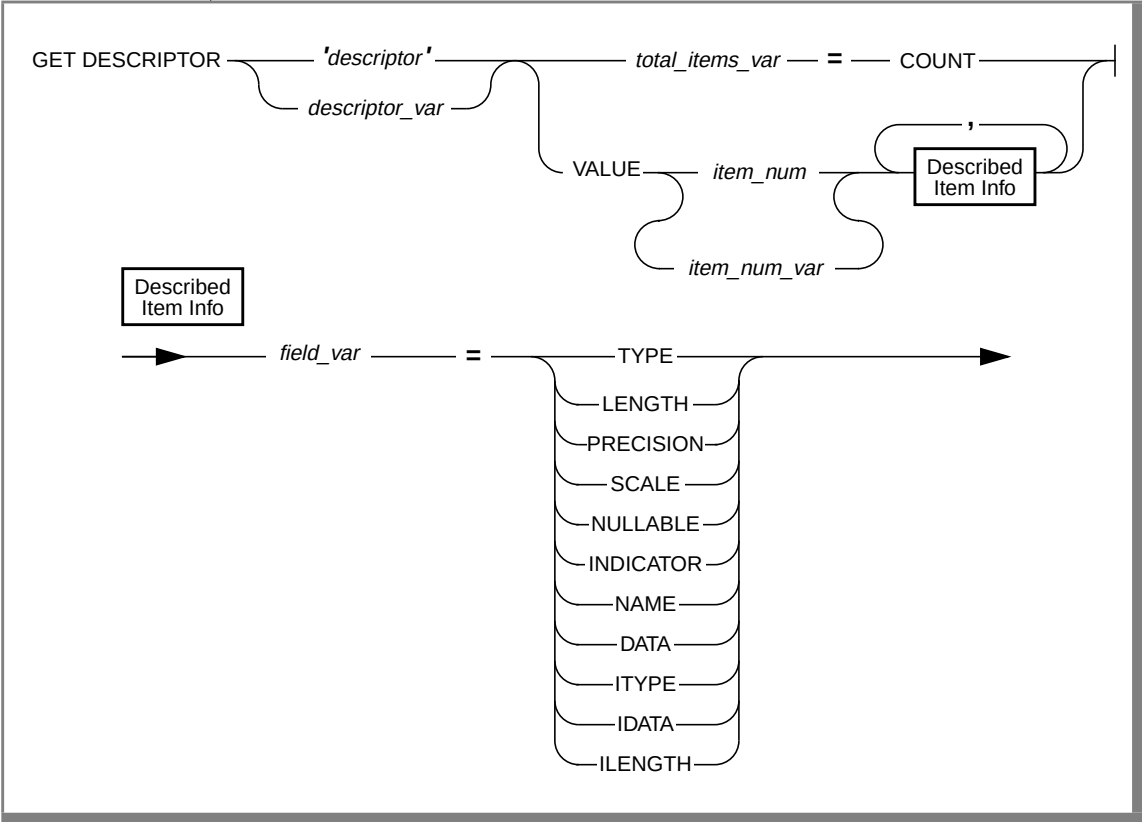
GET DESCRIPTOR

Use the GET DESCRIPTOR statement to accomplish the following separate tasks:

- Determine how many values are described in a system-descriptor area
- Determine the characteristics of each column or expression that is described in the system-descriptor area
- Copy a value from the system-descriptor area into a host variable after a FETCH statement

Use this statement with ESQL/C.

Syntax



GET DESCRIPTOR

Element	Purpose	Restrictions	Syntax
<i>descriptor</i>	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 <code>ALLOCATE DESCRIPTOR</code> statement.	Quoted String, p. 4-157
<i>descriptor_var</i>	Embedded variable name that holds the value of <i>descriptor</i>	The system-descriptor area identified in <i>descriptor_var</i> must have been allocated in an <code>ALLOCATE DESCRIPTOR</code> statement.	Name must conform to language-specific rules for variable names.
<i>field_var</i>	Host variable that receives the contents of the specified field from the system-descriptor area	The <i>field_var</i> must be an appropriate type to receive the value of the specified field from the system-descriptor area.	Name must conform to language-specific rules for variable names.
<i>item_num</i>	Unsigned integer that represents one of the items described in the system-descriptor area	The value of <i>item_num</i> must be greater than zero and less than the number of item descriptors that were specified when the system-descriptor area was allocated with the <code>ALLOCATE DESCRIPTOR</code> statement.	Literal Number, p. 4-139
<i>item_num_var</i>	Host variable that holds the value of <i>item_num</i>	The <i>item_num_var</i> must be an integer data type.	Name must conform to language-specific rules for variable names.
<i>total_items_var</i>	Host variable that indicates how many items are described in the system-descriptor area	The host variable must be an integer data type.	Name must conform to language-specific rules for variable names.

Usage

If an error occurs during the assignment to any identified host variable, the contents of the host variable are undefined.

The GET DESCRIPTOR statement can be used in EXECUTE PROCEDURE statements, which have been described with the USING SQL DESCRIPTOR parameter.

The host variables that are used in the GET DESCRIPTOR statement must be declared in the BEGIN DECLARE SECTION of a program.

Using the COUNT Keyword

Use the COUNT keyword to determine how many items are described in the system-descriptor area.

The following ESQL/C example shows how to use a GET DESCRIPTOR statement with a host variable to determine how many items are described in the system-descriptor area called **desc1**:

```
main()
{
EXEC SQL BEGIN DECLARE SECTION;
int h_count;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate descriptor 'desc1' with max 20;

/* This section of program would prepare a SELECT or INSERT
 * statement into the s_id statement id.
 */
EXEC SQL describe s_id using sql descriptor 'desc1';

EXEC SQL get descriptor 'desc1' :h_count = count;
...
}
```


Using the VALUE Clause

Use the VALUE clause to obtain information about a described column or expression or to retrieve values that the database server returns.

The *item_num* must be greater than zero and less than the number of item descriptors that were specified when the system-descriptor area was allocated with ALLOCATE DESCRIPTOR.

Using the VALUE Clause After a DESCRIBE

After you describe a SELECT, EXECUTE PROCEDURE, INSERT, or UPDATE statement, the characteristics of each column or expression in the select list of the SELECT statement, the characteristics of the values returned by the EXECUTE PROCEDURE statement, or the characteristics of each column in a INSERT or UPDATE statement are returned to the system-descriptor area. Each value in the system-descriptor area describes the characteristics of one returned column or expression.

The following ESQL/C example shows how to use a GET DESCRIPTOR statement to obtain data type information from the **demodesc** system-descriptor area:

```
EXEC SQL get descriptor 'demodesc' value :index
      :type = TYPE,
      :len = LENGTH,
      :name = NAME;
printf("Column %d: type = %d, len = %d, name = %s\n",
      index, type, len, name);
```

The value that the database server returns into the TYPE field is a defined integer. To evaluate the data type that is returned, test for a specific integer value. For additional information about integer data type values, see [page 2-548](#).

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, IDATA, or ITYPE is used. It indicates that these fields are not standard X/Open fields for a system-descriptor area. ♦

If the **TYPE** of a fetched value is **DECIMAL** or **MONEY**, the database server returns the precision and scale information for a column into the **PRECISION** and **SCALE** fields after a **DESCRIBE** statement is executed. If the **TYPE** is *not* **DECIMAL** or **MONEY**, the **SCALE** and **PRECISION** fields are undefined.

Using the VALUE Clause After a FETCH

Each time your program fetches a row, it must copy the fetched value into host variables so that the data can be used. To accomplish this task, use a **GET DESCRIPTOR** statement after each fetch of each value in the select list. If three values exist in the select list, you need to use three **GET DESCRIPTOR** statements after each fetch (assuming you want to read all three values). The item numbers for each of the three **GET DESCRIPTOR** statements are 1, 2, and 3.

The following **ESQL/C** example shows how you can copy data from the **DATA** field into a host variable (**result**) after a fetch. For this example, it is predetermined that all returned values are the same data type.

```
EXEC SQL get descriptor 'demodesc' :desc_count = count;
.
.
.
EXEC SQL fetch democursor using sql descriptor 'demodesc';
for (i = 1; i <= desc_count; i++)
{
    if (sqlca.sqlcode != 0) break;
    EXEC SQL get descriptor 'demodesc' value :i :result = DATA;
    printf("%s ", result);
}
printf("\n");
```

Fetching a Null Value

When you use **GET DESCRIPTOR** after a fetch, and the fetched value is null, the **INDICATOR** field is set to -1 (null). The value of **DATA** is undefined if **INDICATOR** indicates a null value. The host variable into which **DATA** is copied has an unpredictable value.

Using LENGTH or ILENGTH

If your **DATA** or **IDATA** field contains a character string, you must specify a value for **LENGTH**. If you specify **LENGTH=0**, **LENGTH** is automatically set to the maximum length of the string. The **DATA** or **IDATA** field might contain a literal character string or a character string that is derived from a character variable of **CHAR** or **VARCHAR** data type. This provides a method to determine the length of a string in the **DATA** or **IDATA** field dynamically.

If a **DESCRIBE** statement precedes a **GET DESCRIPTOR** statement, **LENGTH** is automatically set to the maximum length of the character field that is specified in your table.

This information is identical for **ILENGTH**. Use **ILENGTH** when you create a dynamic program that does not comply with the X/Open standard.

References

Related statements: **ALLOCATE DESCRIPTOR**, **DEALLOCATE DESCRIPTOR**, **DECLARE**, **DESCRIBE**, **EXECUTE**, **FETCH**, **OPEN**, **PREPARE**, **PUT**, **SET DEFERRED_PREPARE**, and **SET DESCRIPTOR**

For more information on concepts relating to the **GET DESCRIPTOR** statement, see the [*INFORMIX-ESQL/C Programmer's Manual*](#).

+

E/C

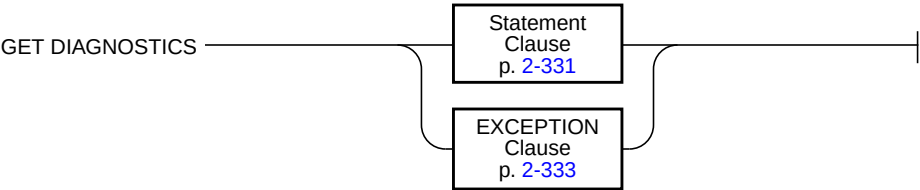
GET DIAGNOSTICS

Use the GET DIAGNOSTICS statement to return diagnostic information about executing an SQL statement. The GET DIAGNOSTICS statement uses one of two clauses, as described in the following list:

- 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.

Use this statement with ESQL / C.

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 GET DIAGNOSTICS statement never changes the contents of the diagnostics area.

Using the SQLSTATE Error Status Code

When an SQL statement executes, an error status code is automatically generated. This code represents success, failure, warning, or no data found. This error status code is stored in a variable called **SQLSTATE**.

Class and Subclass Codes

THE **SQLSTATE** status code is a five-character string that can contain only digits and capital letters.

The first two characters of the **SQLSTATE** status code indicate a class. The last three characters of the **SQLSTATE** code indicate a subclass. Figure 2-1 shows the structure of the **SQLSTATE** code. This example uses the value 08001, where 08 is the class code and 001 is the subclass code. The value 08001 represents the error unable to connect with database environment.

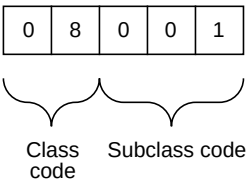


Figure 2-1
Structure of the SQLSTATE Code

The following table is a quick reference for interpreting class code values.

SQLSTATE Class Code Value	Outcome
00	Success
01	Success with warning
02	No data found
> 02	Error or warning

Support for 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 an I are Informix-specific warning messages.
- **SQLSTATE** codes with a class code of IX and any subclass code are Informix-specific error messages.
- **SQLSTATE** codes whose class code begins with a digit in the range 5 to 9 or with 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.

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** error status code.

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

(1 of 4)

Class	Subclass	Meaning
01	I01	Database has transactions
01	I03	ANSI-compliant database selected
01	I04	Informix Dynamic 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
02	000	No data found
07	000	Dynamic SQL error
07	001	USING clause does not match dynamic parameters
07	002	USING clause does not match target specifications
07	003	Cursor specification cannot be executed
07	004	USING clause is required for dynamic parameters
07	005	Prepared statement is not a cursor specification
07	006	Restricted data type attribute violation
07	008	Invalid descriptor count
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

(2 of 4)

Class	Subclass	Meaning
0A	000	Feature not supported
0A	001	Multiple server transactions
21	000	Cardinality violation
21	S01	Insert value list does not match column list
21	S02	Degree of derived table does not match column list
22	000	Data exception
22	001	String data, right truncation
22	002	Null value, no indicator parameter
22	003	Numeric value out of range
22	005	Error in assignment
22	027	Data exception trim error
22	012	Division by zero
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

(3 of 4)

Class	Subclass	Meaning
37	000	Syntax error or access violation in PREPARE or EXECUTE IMMEDIATE
3C	000	Duplicate cursor name
40	000	Transaction rollback
40	003	Statement completion unknown
42	000	Syntax error or access violation
S0	000	Invalid name
S0	001	Base table or view table already exists
S0	002	Base table not found
S0	011	Index already exists
S0	021	Column already exists
S1	001	Memory allocation failure
IX	000	Informix reserved error message

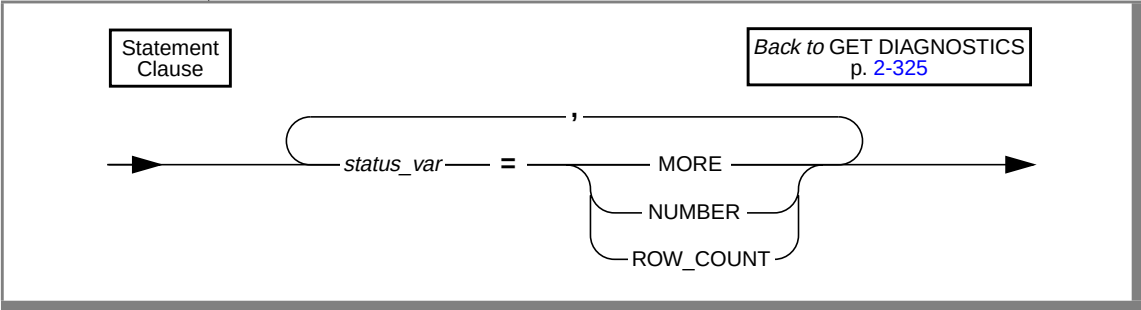
(4 of 4)

Using SQLSTATE in Applications

You can use a variable, called **SQLSTATE**, that you do not have to declare in your program. **SQLSTATE** contains the error code that is essential for error handling, which is generated every time your program executes an SQL statement. **SQLSTATE** is created automatically. You can examine the **SQLSTATE** variable to determine whether an SQL statement was successful. If the **SQLSTATE** variable indicates that the statement failed, you can execute a GET DIAGNOSTICS statement to obtain additional error information.

For an example of how to use an **SQLSTATE** variable in a program, see [“Using GET DIAGNOSTICS for Error Checking” on page 2-340](#).

Statement Clause



Element	Purpose	Restrictions	Syntax
status_var	Host variable that receives status information about the most recent SQL statement Receives information for the specified status field name.	Data type must match that of the requested field.	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 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

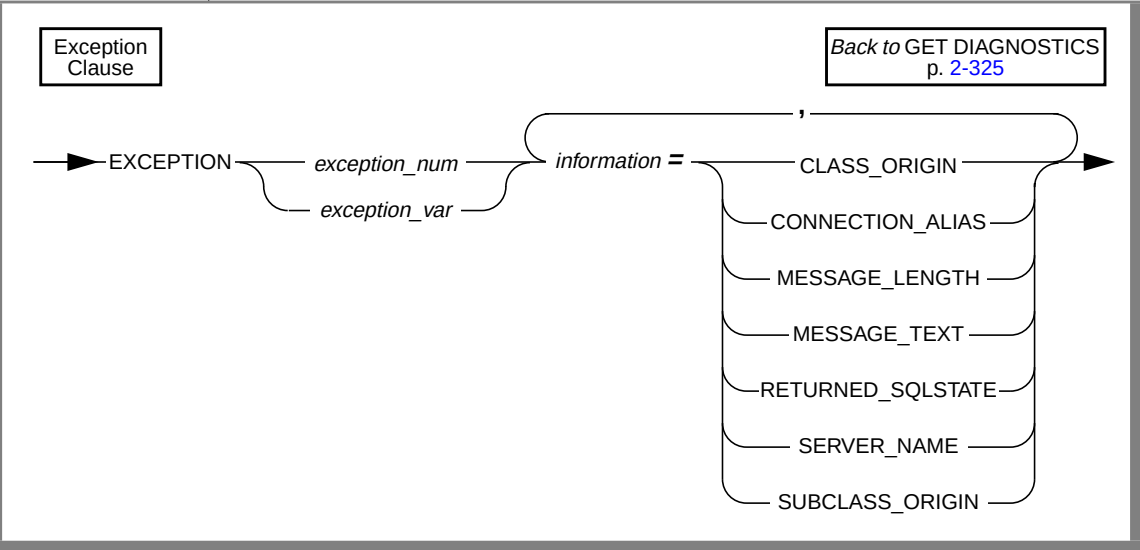
Use the NUMBER keyword to count the number of exceptions that the most recently executed SQL statement placed into the diagnostics area. The **NUMBER** field can hold a value from 1 to 35,000, depending on how many exceptions are counted.

Using the ROW_COUNT Keyword

Use the ROW_COUNT keyword to count the number of rows the most recently executed statement processed. ROW_COUNT counts the following number of rows:

- Inserted into a table
- Updated in a table
- Deleted from a table

EXCEPTION Clause



Element	Purpose	Restrictions	Syntax
<i>exception_num</i>	Literal integer value that specifies the exception number for a GET DIAGNOSTICS statement The <i>exception_num</i> 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. 4-139
<i>exception_var</i>	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.
<i>information</i>	Host variable that 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 retrieving exception information, GET DIAGNOSTICS 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]
MESSAGE_TEXT	Character	String	char[255]
MESSAGE_LENGTH	Integer	Numeric value	int
SERVER_NAME	Character	String	char[255]
CONNECTION_NAME	Character	String	char[255]

The application specifies the exception by number, using either an unsigned integer or an integer host variable (an exact numeric with a scale of 0). An exception with a value of 1 corresponds to the **SQLSTATE** value set by the most recent SQL statement other than GET DIAGNOSTICS. The association between other exception numbers and other exceptions raised by that SQL statement is undefined. Thus, no set order exists in which the diagnostic area can be filled with exception values. You always get at least one exception, even if the **SQLSTATE** value indicates success.

If an error occurs within the GET DIAGNOSTICS statement (that is, if an 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 CLASS_ORIGIN Keyword

Use the CLASS_ORIGIN keyword to retrieve the class portion of the **RETURNED_SQLSTATE** value. If the International Standards Organization (ISO) standard defines the class, the value of CLASS_ORIGIN is equal to ISO 9075. Otherwise, the value of CLASS_ORIGIN is defined by Informix and cannot be ISO 9075. ANSI SQL and ISO SQL are synonymous.

Using the SUBCLASS_ORIGIN Keyword

Use the SUBCLASS_ORIGIN keyword to define the source of the subclass portion of the **RETURNED_SQLSTATE** value. If the ISO international standard defines the subclass, the value of SUBCLASS_ORIGIN is equal to ISO 9075.

Using the MESSAGE_TEXT Keyword

Use the MESSAGE_TEXT keyword to determine the message text of the exception (for example, an error message).

Using the MESSAGE_LENGTH Keyword

Use the MESSAGE_LENGTH keyword to determine the length of the current MESSAGE_TEXT string.

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	Contains the name of the database server to which you connect or fail to connect. Field is blank if you do not have a current connection or if you make a default connection.
SET CONNECTION	Contains the name of the database server to which you switch or fail to switch
DISCONNECT	Contains the name of the database server from which you disconnect or fail to disconnect If you disconnect and then you execute a DISCONNECT statement for a connection that is not current, the SERVER_NAME field remains unchanged.
DISCONNECT ALL	Sets the field to blank if the statement executes successfully If the statement 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	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	Contains the name of the connection, specified in the CONNECT statement, to which you switch or fail to switch
DISCONNECT	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	Contains no information if the statement executes successfully If the statement does not execute successfully, the CONNECTION_NAME field contains the names of all the connections, specified in your CONNECT statement, from which you did not disconnect. However, this information does not mean that the connection still exists.

If 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

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 example illustrates how to use the GET DIAGNOSTICS statement to display error information. The example shows an ESQL/C error display routine called **disp_sqlstate_err()**.

```
void disp_sqlstate_err()
{
    int j;

    EXEC SQL BEGIN DECLARE SECTION;
        int exception_count;
        char overflow[2];
        int exception_num=1;
        char class_id[255];
        char subclass_id[255];
        char message[255];
        int messlen;
        char sqlstate_code[6];
        int i;
    EXEC SQL END DECLARE SECTION;

    printf("-----");
    printf("-----\n");
    printf("SQLSTATE: %s\n",SQLSTATE);
    printf("SQLCODE: %d\n", SQLCODE);
    printf("\n");

    EXEC SQL get diagnostics :exception_count = NUMBER,
        :overflow = MORE;
    printf("EXCEPTIONS:  Number=%d\t", exception_count);
    printf("More? %s\n", overflow);
    for (i = 1; i <= exception_count; i++)
```

```

{
    EXEC SQL get diagnostics exception :i
        :sqlstate_code = RETURNED_SQLSTATE,
        :class_id = CLASS_ORIGIN, :subclass_id = SUBCLASS_ORIGIN,
        :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
    printf("- - - - -\n");
    printf("EXCEPTION %d: SQLSTATE=%s\n", i,
        sqlstate_code);
    message[messlen-1] = '\0';
    printf("MESSAGE TEXT: %s\n", message);

    j = stleng(class_id);
    while((class_id[j] == '\0') ||
        (class_id[j] == ' '))
        j--;
    class_id[j+1] = '\0';
    printf("CLASS ORIGIN: %s\n", class_id);

    j = stleng(subclass_id);
    while((subclass_id[j] == '\0') ||
        (subclass_id[j] == ' '))
        j--;
    subclass_id[j+1] = '\0';
    printf("SUBCLASS ORIGIN: %s\n", subclass_id);
}

printf("-----");
printf("-----\n");
}

```

References

For a task-oriented discussion of error handling and the **SQLSTATE** variable, see the [Informix Guide to SQL: Tutorial](#).

For a discussion of concepts related to the GET DIAGNOSTICS statement and the **SQLSTATE** variable, see the [INFORMIX-ESQL/C Programmer's Manual](#).

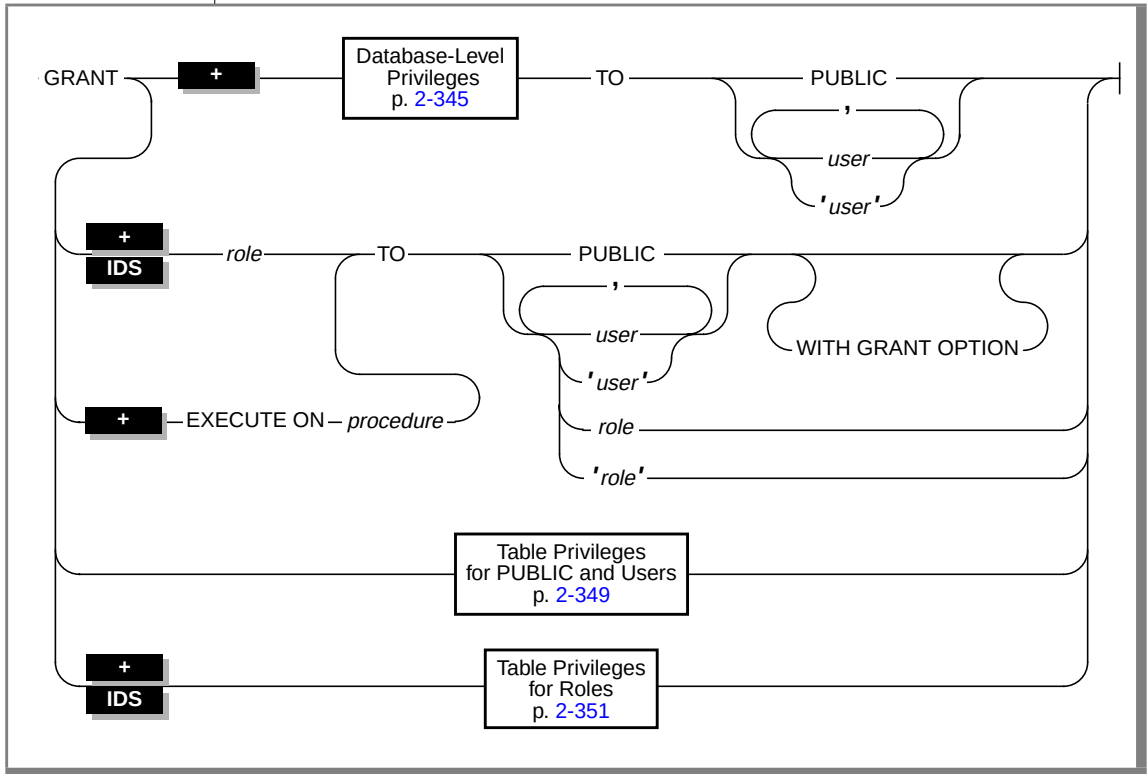
GRANT

Use the GRANT statement to grant the following privileges:

- Privileges on a database to a user
- Privileges on a table, view, or synonym to a user or a role
- Privileges on a procedure to a user or a role

You can also use GRANT to grant a role to a user or to another role.

Syntax



Element	Purpose	Restrictions	Syntax
<i>procedure</i>	Name of the stored procedure that the specified user or role can execute	The stored procedure must be owned by the user who issues the GRANT statement.	Database Object Name, p. 4-25
<i>role</i>	Name of the role that is granted, or the name of the role to which another role is granted	The role must have been created with the CREATE ROLE statement.	Identifier, p. 4-113
<i>user</i>	Name of the user to whom a role is granted, or the name of a user who receives the specified privilege	If you enclose <i>user</i> in quotation marks, 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 <i>user</i>. If you grant a privilege to PUBLIC, you do not need to grant the privilege to individual users because PUBLIC extends the privilege to all authorized users.	Identifier, p. 4-113

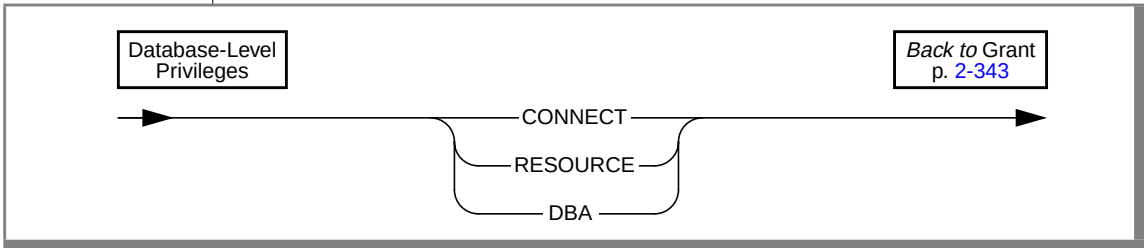
Usage

A GRANT statement can extend user privileges but cannot limit existing privileges. Later GRANT statements do not affect privileges already granted to a user. When database-level privileges collide with table-level privileges, the more-restrictive privileges take precedence. You can grant table-level privileges on a table or on a view.

Privileges granted to users remain in effect until you cancel them with a REVOKE statement. Only grantors can revoke the privileges that they previously issued.

You can grant privileges to a role, and you can grant a role to individual users or to another role. For further information, see [“Granting a Role to a User or Another Role” on page 2-348](#).

Database-Level Privileges



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.

Three levels of database privileges control access. These privilege levels are, from lowest to highest, Connect, Resource, and DBA. These privileges are associated with the following keywords.

Privilege	Functions
CONNECT	Gives you the ability to query and modify data. You can modify the database schema if you own the database object you want to modify. Any user with the Connect privilege can perform the following functions: ■ Connect to the database with the CONNECT statement or another connection statement ■ Execute SELECT, INSERT, UPDATE, and DELETE statements, provided the user has the necessary table-level privileges ■ Create views, provided the user has the Select privilege on the underlying tables ■ Create synonyms ■ Create temporary tables and create indexes on the temporary tables ■ Alter or drop a table or an index, provided the user owns the table or index (or has Alter, Index, or References privileges on the table) ■ Grant privileges on a table or view, provided the user owns the table (or has been given privileges on the table with the WITH GRANT OPTION keyword)
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 functions: ■ Create new tables ■ Create new indexes ■ Create new procedures

(1 of 2)

Privilege	Functions
DBA	Has all the capabilities of the Resource privilege as well as the ability to perform the following functions: ■ 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 ■ Insert, delete, or update rows of any system catalog table except systables ■ Drop any database object, regardless of its owner ■ Create tables, views, and indexes, and specify another user as owner of the database objects ■ Execute the DROP DATABASE statement ■ Execute the DROP DISTRIBUTIONS option of the UPDATE STATISTICS statement

(2 of 2)

User **informix** has the privilege required to alter tables in the system catalog, including the **systables** table.



Warning: Although the user **informix** and DBAs can modify most system catalog tables (only user **informix** can modify **systables**), Informix strongly recommends that you do not update, delete, or alter any rows in them. Modifying the system catalog tables can destroy the integrity of the database.

The following example uses the PUBLIC keyword to grant the Connect privilege on the currently active database to all users:

```
GRANT CONNECT TO PUBLIC
```

Granting a Role to a User or Another Role

You can use the GRANT statement to grant a role to another role or user. You can only grant roles that have been created with the CREATE ROLE statement. For an explanation of roles, see the CREATE ROLE statement on [page 2-146](#).

After a role is granted, you must use the SET ROLE statement to enable the role. Users who have been granted a role with the WITH GRANT OPTION can grant that role to other users or roles. Roles that are granted to users remain granted until a REVOKE statement cancels them.

Table-level privileges and the Execute privilege to stored procedures can be granted to roles. Database-level privileges cannot be granted to roles.

The DBA or a user who is granted the role with the WITH GRANT OPTION can grant a role to a user or to another role. A role cannot be granted to itself, either directly or indirectly. The following statement causes an error:

```
GRANT engineer TO engineer -- Causes an error
```

The following example grants the role **engineer** to the user **maryf**:

```
GRANT engineer TO maryf
```

The following example grants the role **acct** to the role **engineer**:

```
GRANT acct TO engineer
```

The following example grants the role **engineer** with the WITH GRANT OPTION to the user **maryf**. This privilege allows **maryf** to grant the role **engineer** to other users or roles.

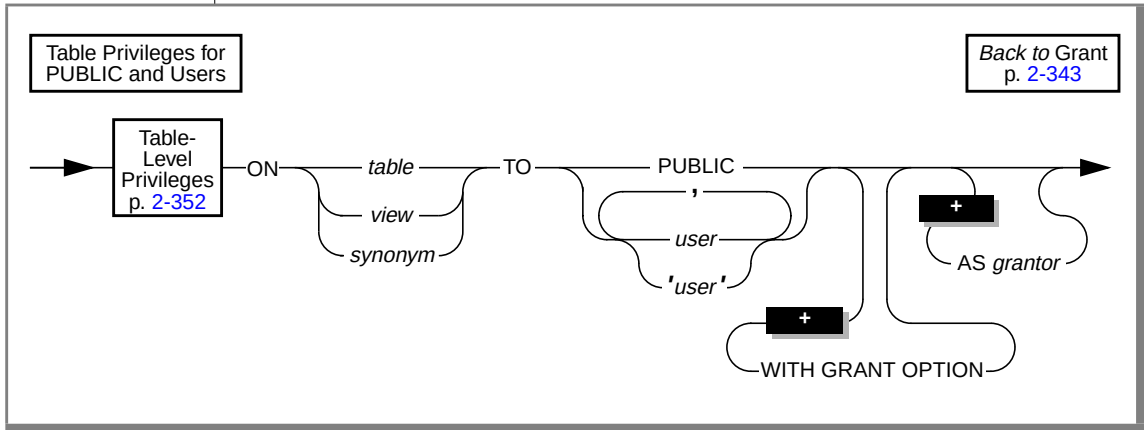
```
GRANT engineer TO maryf WITH GRANT OPTION
```

Stored Procedure Privileges

Use the EXECUTE ON option with a procedure name to grant another user or a role the ability to run a stored procedure that you own.

When you create an owner-privileged stored procedure, the default privilege is PUBLIC.

Table Privileges for PUBLIC and Users



Element	Purpose	Restrictions	Syntax
<i>grantor</i>	Name of the person who is to be listed as the source of the specified privilege in the sysdbaauth system catalog table The person who issues the GRANT statement is the default grantor of the privilege.	If you specify someone else as the grantor of the specified privilege to <i>user</i> , you cannot later revoke that privilege from <i>user</i> .	Identifier, p. 4-113
<i>synonym</i>	Name of the synonym on which privileges are granted	The synonym and the table to which the synonym points must reside in the current database.	Database Object Name, p. 4-25
<i>table</i>	Name of the table on which privileges are granted	The table must reside in the current database.	Database Object Name, p. 4-25

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>user</i>	Name of the user who receives the specified privilege	If you enclose <i>user</i> in quotation marks, 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 <i>user</i>. If you grant a privilege to PUBLIC, you do not need to grant the privilege to individual users because PUBLIC extends the privilege to all authorized users. See also “Restricting Privileges at the Table Level” on page 2-355.	Identifier, p. 4-113
<i>view</i>	Name of the view on which privileges are granted	The view must reside in the current database.	Database Object Name, p. 4-25

(2 of 2)

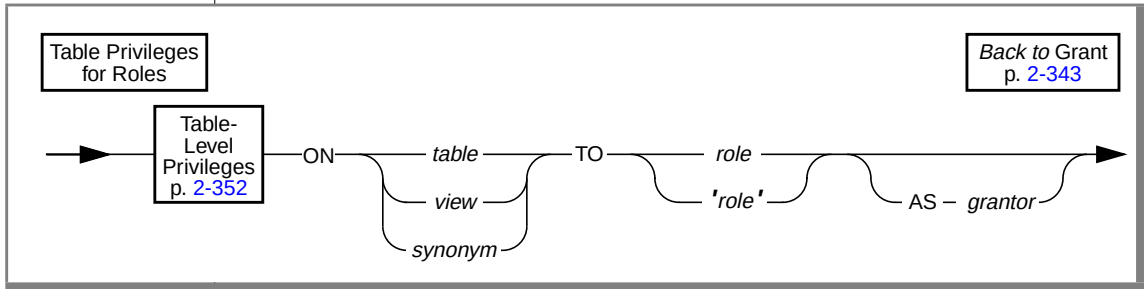
You can grant privileges on a table, view, or synonym to a user, a list of users, or all users (PUBLIC). When you grant these privileges to users or PUBLIC, you can also specify the WITH GRANT OPTION clause and the AS *grantor* clause.

The table, view, or synonym on which you grant privileges must reside in the current database.

The following example grants the table-level privilege Insert on **table1** to the user named **mary**:

```
GRANT INSERT ON table1 TO mary
```

Table Privileges for Roles



Element	Purpose	Restrictions	Syntax
<i>grantor</i>	Name of the person who is to be listed as the source of the specified privilege in the sysdba system catalog table. The person who issues the GRANT statement is the default grantor of the privilege.	If you specify someone else as the grantor of the specified privilege to <i>role</i> , you cannot later revoke that privilege from <i>role</i> .	Identifier, p. 4-113
<i>role</i>	Name of the role to which the specified privilege is granted	The role must have been created with the CREATE ROLE statement.	Identifier, p. 4-113
<i>synonym</i>	Name of the synonym on which privileges are granted	The synonym and the table to which the synonym points must reside in the current database.	Database Object Name, p. 4-25
<i>table</i>	Name of the table on which privileges are granted	The table must reside in the current database.	Database Object Name, p. 4-25
<i>view</i>	Name of the view on which privileges are granted	The view must reside in the current database.	Database Object Name, p. 4-25

You can grant privileges on a table, view, or synonym to a role. When you grant these privileges to a role, you can also specify the AS *grantor* clause, but you cannot specify the WITH GRANT OPTION clause.

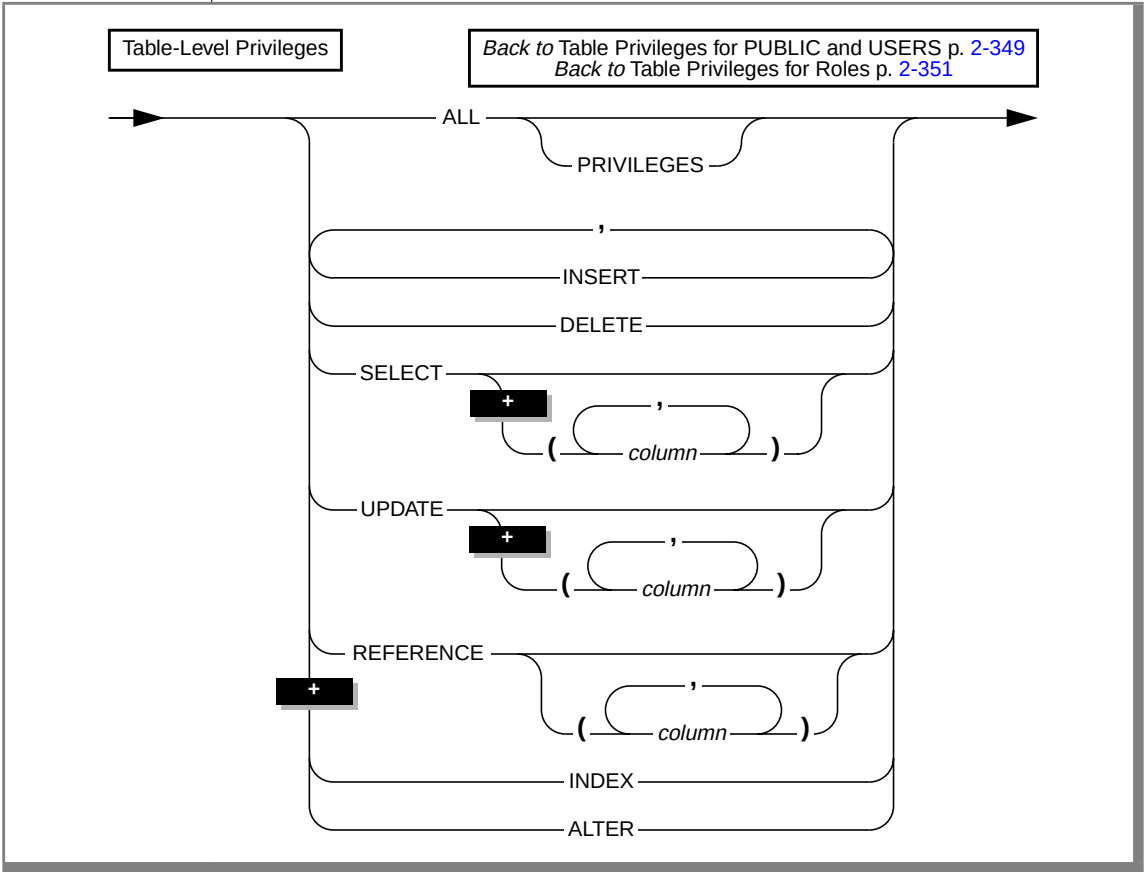
The table, view, or synonym on which you grant privileges must reside in the current database.

The following example grants the table-level privilege Insert on **table1** to the role **engineer**:

```
GRANT INSERT ON table1 TO engineer
```

Table-Level Privileges

The table-level privileges clause lets you control privileges at the table level.



Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of the column or columns to which a Select, Update, or References privilege is restricted If you omit <i>column</i> , the privilege applies to all columns in the specified table.	The specified column or columns must exist.	Identifier, p. 4-113

As the owner of a table, or as the DBA, you control access to the table through seven table-level privileges. Four privileges control access to the table data: Select, Insert, Delete, and Update. The remaining three privileges are Index, which controls index creation; Alter, which controls the ability to change the table definition or alter an index; and References, which controls the ability to place referential constraints on table columns.

The person who creates a table is its owner and receives all seven table-level privileges. Table ownership cannot be transferred to another user.

To use the GRANT statement, list the privileges that you are granting to *user*. If you are granting all table-level privileges, use the keyword ALL. If you are granting the Select, Update, or References privilege, you can limit the privileges by listing the names of specific columns.

If you are granting the Alter privilege with the intent of allowing a user to make changes to a table, you must also grant the Resource privilege for the database in which the table resides.

If you are granting the Index privilege with the intent of allowing *user* to make changes to the underlying structure of a table, be aware that *user* must also have the Resource privilege for the database to be able to modify the database structure. The table-level privileges are defined in the following table.

Privilege	Functions
INSERT	Provides the ability to insert rows
DELETE	Provides the ability to delete rows
SELECT	Provides the ability to name any column in SELECT statements You can restrict the Select privilege to one or more columns by listing them.
UPDATE	Provides the ability to name any column in UPDATE statements You can restrict the Update privilege to one or more columns by listing them.

(1 of 2)

Privilege	Functions
REFERENCES	Provides the ability to reference columns in referential constraints You must have the Resource privilege to take advantage of the References privilege. (However, you can add a referential constraint during an ALTER TABLE statement. This action does not require that you have the Resource privilege on the database.) You can restrict the References privilege to one or more columns by listing them. You need only the References privilege to indicate cascading deletes. You do not need the Delete privilege to place cascading deletes on a table.
INDEX	Provides the ability to create permanent indexes You must have Resource privilege to use the Index privilege. (Any user with the Connect privilege can create an index on temporary tables.)
ALTER	Provides the ability to add or delete columns, modify column data types, or add or delete constraints This privilege also provides the ability to set the database object mode of unique indexes and constraints to the enabled, disabled, or filtering mode. In addition, this privilege provides the ability to set the database object mode of nonunique indexes and triggers to the enabled or disabled modes. You must have Resource privilege to use the Alter privilege.
ALL	Provides all privileges The PRIVILEGES keyword is optional.

(2 of 2)

The following example grants, to users **mary** and **john**, the Delete and Select privileges on all columns. It also grants the Update privilege on **customer_num**, **fname**, and **lname** for the **customer** table.

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
ON customer TO mary, john
```

To grant these table-level privileges 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
```

Restricting Privileges at the Table Level

You must take action to restrict privileges at the table level. The database server automatically grants to PUBLIC all table-level privileges, except Alter and References, when you create a table. To limit table access, you must revoke all privileges and regrant only those you want, as the following example shows:

```
REVOKE ALL ON customer FROM PUBLIC
GRANT ALL ON customer TO john, mary
GRANT SELECT (fname, lname, company, city)
ON customer TO PUBLIC
```

ANSI

In an ANSI-compliant database, only the table owner receives privileges when a table is created. ♦

Behavior of the ALL Keyword

The ALL keyword grants all table-level privileges to the specified user. If any or all of the table-level privileges do not exist for the grantor, the GRANT statement with the ALL keyword succeeds, 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 seven 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. The SQLSTATE warning is returned with a successful statement for the following reasons:

- The statement succeeds in granting the Select AND Insert privileges to user **tania** because user **ted** has those privileges and the right to grant those privileges to other users.
- SQLSTATE code 01007 is returned because the other five privileges implied by the ALL keyword (the Delete, Update, References, Index, and Alter privileges) were not grantable by user **ted** and, therefore, were not granted to user **tania**.

WITH GRANT OPTION Keywords

Using the WITH GRANT OPTION keyword conveys the specified privilege to *user* along with the right to grant those same privileges to other users. You create a chain of privileges that begins with you and extends to *user* as well as to whomever *user* conveys the right to grant privileges. If you use the WITH GRANT OPTION keyword, you can no longer control the dissemination of privileges.

If you revoke from *user* the privilege that you granted using the WITH GRANT OPTION keyword, you sever the chain of privileges. That is, when you revoke privileges from *user*, you automatically revoke the privileges of all users who received privileges from *user* or from the chain that *user* created (unless *user*, or the users who received privileges from *user*, were granted the same set of privileges by someone else). The following examples illustrate this situation. You, as the owner of the table **items**, issue the following statements to grant access to 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 issue the following statement to cancel access privileges for the user **mary** on the **items** table:

```
REVOKE SELECT, UPDATE ON items FROM mary
```

This single statement effectively revokes all privileges on the **items** table from the users **mary**, **cathy**, and **paul**.

If you want to create a chain of privileges with another user as the source of the privilege, use the AS *grantor* clause.

AS grantor Clause

The AS *grantor* clause lets you establish a chain of privileges with another user as the source of the privileges. This relinquishes your ability to break the chain of privileges. Even a DBA cannot revoke a privilege unless that DBA originally granted the privilege. The following example illustrates this situation. You are the owner of the **items** table, and you grant all privileges to the user **tom**, along with the right to grant all privileges:

```
REVOKE ALL ON items FROM PUBLIC
GRANT ALL ON items TO tom WITH GRANT OPTION
```

The following example illustrates a different situation. You also grant Select and Update privileges to the user **jim**, but you specify that the grant is made as the user **tom**. (The records of the database server show that the user **tom** is the grantor of the grant in the **systabauth** system catalog table, rather than you.)

```
GRANT SELECT, UPDATE ON items TO jim AS tom
```

Later, you decide to revoke privileges on the **items** table from the user **tom**, so you issue the following statement:

```
REVOKE ALL ON items FROM tom
```

When you try to revoke privileges from the user **jim** with a similar statement, however, the database server returns an error, as the following example shows:

```
REVOKE SELECT, UPDATE ON items FROM jim

580: Cannot revoke permission.
```

You get an error because the database-server record shows the original grantor as the user **tom**, and you cannot revoke the privilege. Although you are the table owner, you cannot revoke a privilege that another user granted.

Privileges on a View

You must explicitly grant access privileges on the view to users, because no automatic grant is made to **public**, as is the case with a newly created table.

When you create a view, if you do not own the underlying tables, you must have at least the Select privilege on the table or columns. As view creator, the privileges you have on the underlying table apply to the view built on the table. You do not receive any other privileges or the ability to grant any other privileges because you own the view on the table. If the view meets all the requirements for updating, any Delete, Insert, or Update privileges that you have on the table also apply to 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 cannot grant Index, Alter, or References privileges on a view (or the All privilege because All includes Index, References, and Alter).

For views that reference only tables in the current database, if the owner of a view loses the Select privilege on any table underlying the view, the view is dropped.

For detailed information, refer to the CREATE TABLE statement, which also describes creating views.

References

Related statements: CREATE TABLE, GRANT FRAGMENT, REVOKE, and REVOKE FRAGMENT

For information on roles, see the following statements: CREATE ROLE, DROP ROLE, and SET ROLE.

For a discussion of database-level privileges and table-level privileges, see the *Informix Guide to Database Design and Implementation*.

For a discussion of how to embed GRANT and REVOKE statements in programs, see the [Informix Guide to SQL: Tutorial](#).

+

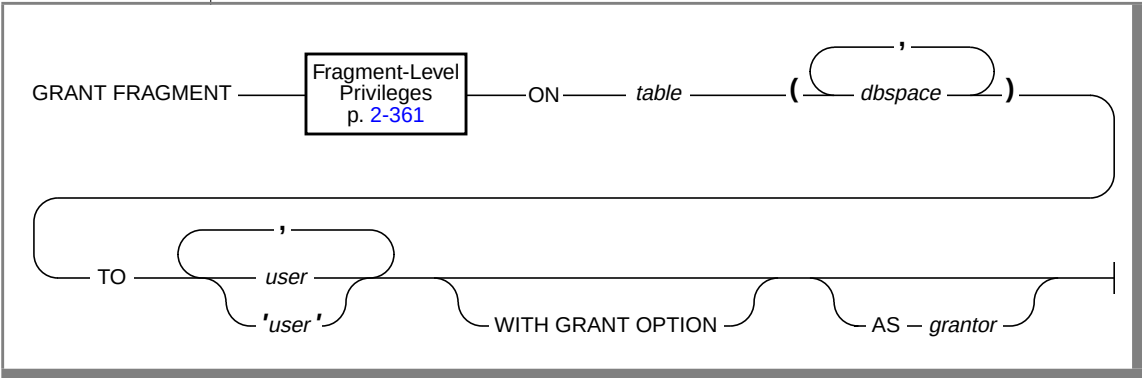
IDS

GRANT FRAGMENT

Use the GRANT FRAGMENT statement to grant Insert, Update, and Delete privileges on individual fragments of a fragmented table.

You can use this statement only with Dynamic Server. This statement is not available with Dynamic Server, Workgroup and Developer Editions.

Syntax



Element	Purpose	Restrictions	Syntax
<i>dbspace</i>	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. No default value exists.	You must specify at least one dbspace. The specified dbspaces must exist.	Identifier, p. 4-113
<i>grantor</i>	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 <i>grantor</i> must be a valid user.	Identifier, p. 4-113

Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of the table that contains the fragment or fragments on which privileges are to be granted No default value exists.	The specified table must exist and must be fragmented by expression.	Database Object Name, p. 4-25
<i>user</i>	Name of the user or users to whom the specified privileges are to be granted No default value exists.	If you enclose <i>user</i> in quotation marks, 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 <i>user</i> .	Identifier, p. 4-113

(2 of 2)

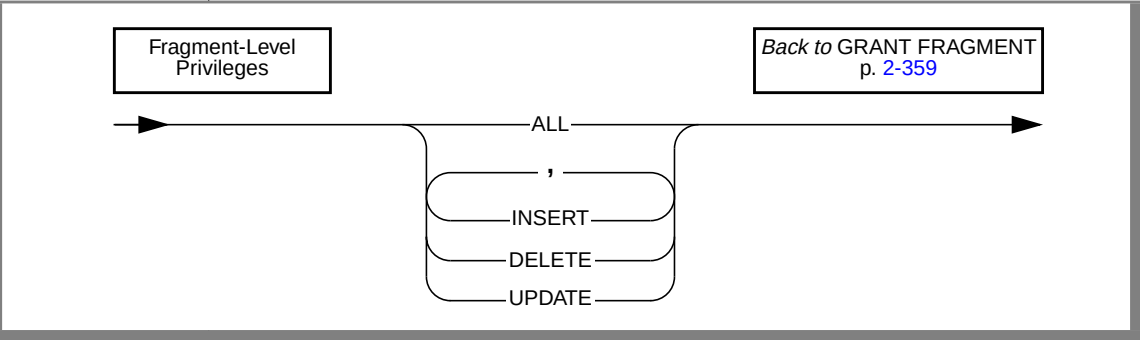
Usage

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 2-8](#).

Fragment-Level Privileges



The following table defines each of the fragment-level privileges.

Privilege	Function
ALL	Provides insert, delete, and update privileges on a fragment
INSERT	Provides the ability to insert rows in the fragment
DELETE	Provides the ability to delete rows in the fragment
UPDATE	Provides 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 the database server validates commands. This algorithm consists of the following checks:

1. When a user executes an INSERT, DELETE, or UPDATE statement, the database server first checks whether the user has the table 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

Related statements: GRANT and REVOKE FRAGMENT

For a discussion of fragment-level and table-level privileges, see the *Informix Guide to Database Design and Implementation*.

+

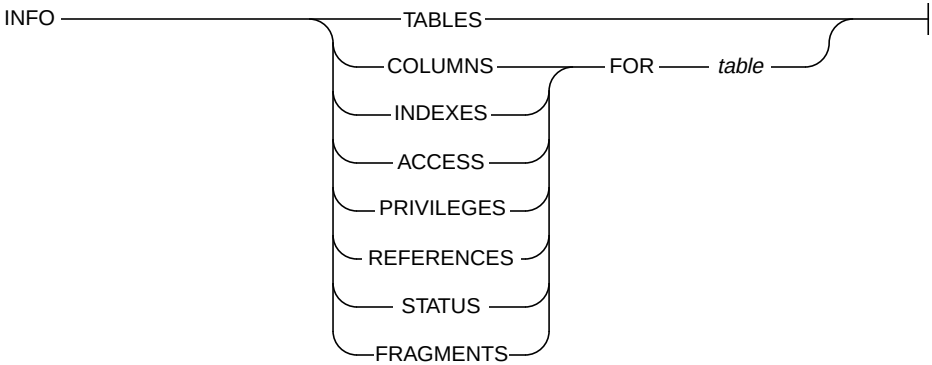
DB

INFO

Use the INFO statement to display a variety of information about databases and tables.

Use this statement with DB-Access.

Syntax



Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of the table for which you want to find information	The table must exist.	Database Object Name, p. 4-25

Usage

You can use the keywords of the INFO statement to display the following types of information:

INFO Keyword	Information Displayed
TABLES	Table names in the current database
COLUMNS	Column information for a specified table
INDEXES	Index information for a specified table
FRAGMENTS	Fragmentation strategy for a specified table
ACCESS	Access privileges for a specified table
PRIVILEGES	Access privileges for a specified table
REFERENCES	References privileges for the columns of a specified table
STATUS	Status information for a specified table

TABLES Keyword

Use the TABLES keyword to display a list of the tables in the current database. The TABLES keyword does not display the system catalog tables.

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**.

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.

INDEXES Keyword

Use the INDEXES keyword to display the name, owner, and type of each index in a specified table, whether the index is clustered, and the names of the columns that are indexed.

FRAGMENTS Keyword

Use the FRAGMENTS keyword to display the dbspace names where fragments are located for a specified table. If the table is fragmented with an expression-based distribution scheme, the INFO statement also shows the expressions.

The FRAGMENTS keyword is not available with Dynamic Server, Workgroup and Developer Editions. ♦

ACCESS Keyword

Use the ACCESS or PRIVILEGES keywords to display user access privileges for a specified table.

REFERENCES Keyword

Use the REFERENCES keyword to display the References privilege for users for the columns of a specified table.

If you want information about database-level privileges, you must use a SELECT statement to access the **sysusers** system catalog table.

STATUS Keyword

Use the STATUS keyword to display information about the owner, row length, number of rows and columns, creation date, and status of audit trails for a specified table.

References

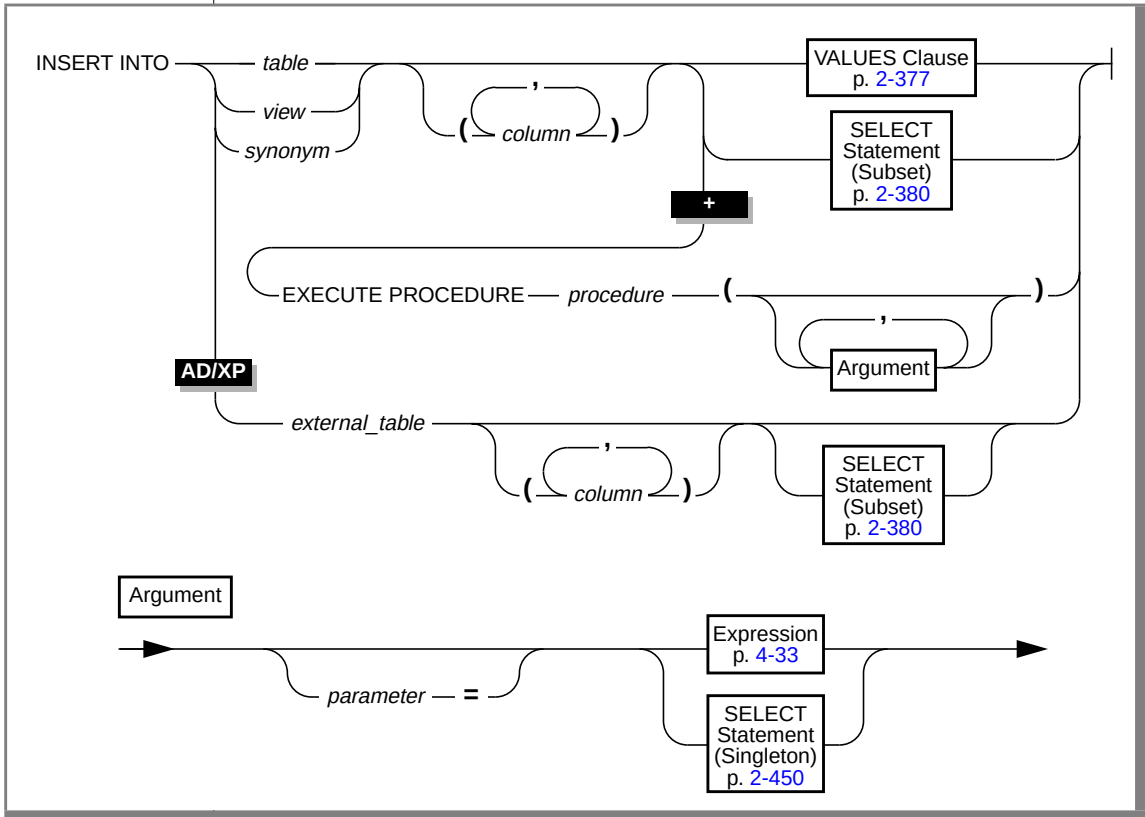
Related statements: GRANT and REVOKE

For a description of the Info option on the SQL menu or the TABLE menu in DB-Access, see the [DB-Access User Manual](#).

INSERT

Use the INSERT statement to insert one or more new rows into a table or view.

Syntax



Element	Purpose	Restrictions	Syntax
<i>column</i>	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. 4-113
<i>external_table</i>	Name of the external table on which you want to insert data	The external table must exist.	Database Object Name, p. 4-25
<i>parameter</i>	Name of an input parameter to the procedure	The input parameter must have been defined in the CREATE PROCEDURE statement for the specified procedure.	Expression, p. 4-33
<i>procedure</i>	Name of the procedure to use to insert the data	The procedure must exist.	Database Object Name, p. 4-25
<i>synonym</i>	Name of the synonym on which you want to insert data	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table on which you want to insert data	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view on which you want to insert data	The view must exist.	Database Object Name, p. 4-25

Usage

Use the INSERT statement to create either a single new row of column values or a group of new rows using data selected from other tables.

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 2-342](#)). 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.”](#)

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.

In ESQL/C, you can use the DESCRIBE statement with an INSERT statement to determine the column order and the data type of the columns in a table. ♦

The number of columns specified in the INSERT INTO clause must equal the number of values supplied in the VALUES clause or by the SELECT statement, either implicitly or explicitly. If you specify 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 2-230](#).

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)
```

E/C

Inserting Rows with a Cursor

In ESQL/C, if you associate a cursor with an INSERT statement, you must use the OPEN, PUT, and CLOSE statements to carry out the INSERT operation. For databases that have transactions but are not ANSI-compliant, you must issue these statements within a transaction.

If you are using a cursor that is associated with an INSERT statement, the rows are buffered before they are written to the disk. The insert buffer is flushed under the following conditions:

- The buffer becomes full.
- A FLUSH statement executes.
- A CLOSE statement closes the cursor.
- In a database that is not ANSI-compliant, an OPEN statement implicitly closes and then reopens the cursor.
- A COMMIT WORK statement ends the transaction.

When the insert buffer is flushed, the client processor performs appropriate data conversion before it sends the rows to the database server. When the database server receives the buffer, it begins to insert the rows one at a time into the database. If an error is encountered while the database server inserts the buffered rows into the database, any buffered rows that follow the last successfully inserted rows are discarded.

Inserting Rows into a Database Without Transactions

If you are inserting rows into a database without transactions, you must take explicit action to restore inserted rows after a failure. For example, if the INSERT statement fails after inserting 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.

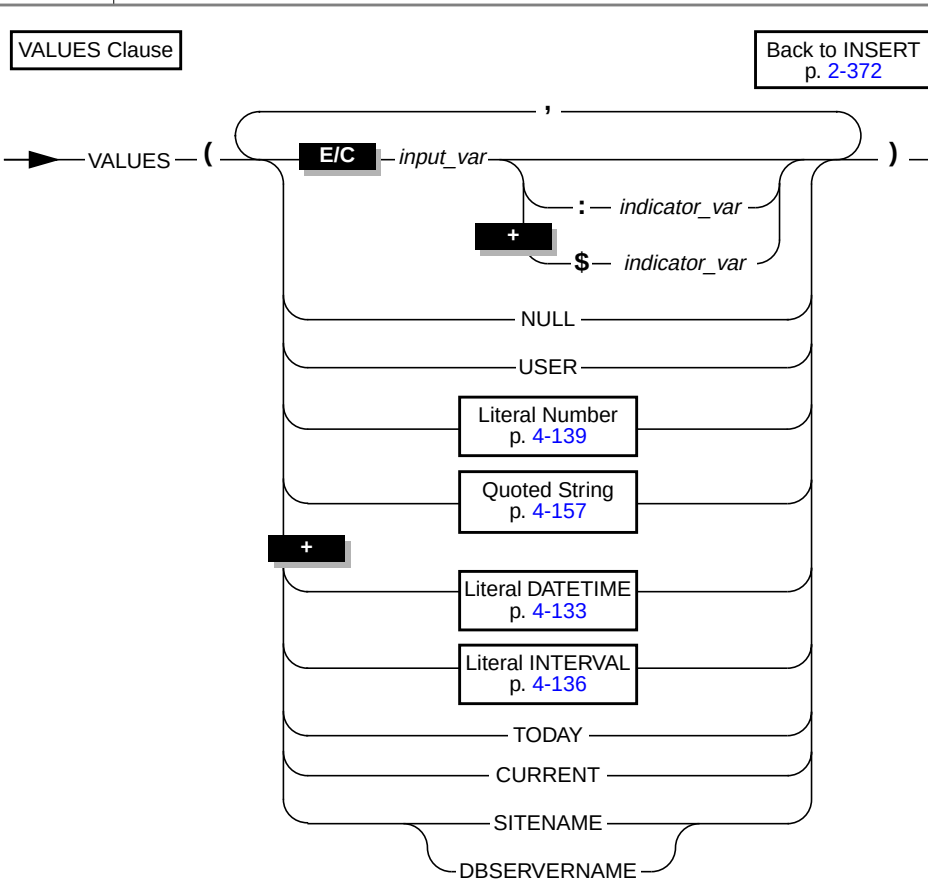
If you are using the database server within an explicit transaction, and the update fails, the database server automatically undoes the effects of the update. ♦

AD/XP

If you are using Dynamic Server with AD and XP Options, tables that you create with the RAW table type are never logged. Thus, RAW tables are not recoverable, even if the database uses logging. For information about RAW tables, refer to the [Informix Guide to SQL: Reference](#). ♦

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
<i>indicator_var</i>	Program variable that indicates when an SQL statement in an ESQL / C program returns a null value to <i>input_var</i>	For restrictions that apply to indicator variables in ESQL / C, see the INFORMIX-ESQL/C Programmer's Manual .	Name must conform to language-specific rules for variable names.
<i>input_var</i>	Host variable that specifies a value to be inserted into a column	You can specify in <i>input_var</i> any other value option listed in the VALUES clause (NULL, Literal Number, and so on).	Name 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 quoted string that is longer than 256 bytes, the database server returns an error.

E/C

In ESQL / C, if you are using variables, you can insert quoted strings longer than 256 bytes into a table. ♦

For discussions on the keywords that you can use in the VALUES clause, refer to [“Constant Expressions” on page 4-47](#).

Value and Column Data Type Compatibility

Although the values you insert do not have to be the same data type as the columns receiving them, the value data type and column data type must be compatible. You can insert only characters into CHAR columns and only numbers or characters that represent number data into number columns. The following example inserts values into the columns of the **customer** table:

```
INSERT INTO customer
VALUES (0, 'Nadia', 'Broadam', 'Ski & Stuff',
      '89 Coniston Road', NULL, 'Short Hills',
      'NJ', '07079', '201-457-4100')
```

The database server makes every effort to perform data conversion. If the data cannot be converted, the INSERT operation fails. Data conversion also fails if the target data type cannot hold the value that is specified. For example, you cannot insert the integer 123456 into a column defined as a SMALLINT data type because this data type cannot hold a number that large.

Inserting Values into SERIAL Columns

If you want to insert consecutive serial values into a SERIAL column in the table, enter a zero for a SERIAL column in the INSERT statement. When a SERIAL column is set to zero, the database server assigns the next highest value. If you want to enter an explicit value into a SERIAL column, specify the nonzero value after you first verify that the value does not duplicate one already in the table. If the SERIAL 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 data type, see the [Informix Guide to SQL: Reference](#).

Using Functions in the VALUES Clause

You can insert the current date, date and time, login name of the current user, or database server name of the current database into a column. 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 DBSERVERNAME or SITENAME function 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 discussions on the keywords that you can use in the VALUES clause, refer to [“Constant Expressions” on page 4-47](#).

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 only if the insert data is selected from another table or tables. In other words, you cannot select data from the table into which you are inserting rows.

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.

Not all clauses and options of the SELECT statement are available for you to use in an INSERT statement. For a detailed discussion of the syntax for the SELECT statement, see [page 2-450](#).

IDS

AD/XP

E/C

The following SELECT clauses and options are not supported:

- FIRST
- ★ INTO TEMP
- ORDER BY
- UNION (in Dynamic Server) ♦

Using External Tables

In Dynamic Server with AD and XP Options, when you create a SELECT statement as a part of a load or unload operation that involves an external table, keep the following restrictions in mind:

- Only one external table is allowed in the FROM clause.
- The SELECT subquery cannot contain an INTO clause, but it can include any valid SQL expression.

When you move data from a database into an external table, the SELECT statement must define all columns in the external table. The SELECT statement must not contain a FIRST, FOR UPDATE, INTO, INTO SCRATCH, or INTO TEMP clause. However, you can use an ORDER BY clause to produce files that are ordered within themselves.

Using INSERT as a Dynamic Management Statement

In ESQL/C, you can use the INSERT statement to handle situations where you need to write code that can insert data whose structure is unknown at the time you compile.

For more information, refer to the dynamic management section of the [*INFORMIX-ESQL/C Programmer's Manual*](#).

Inserting Data Using a Stored Procedure

You can insert the rows of data that result from a procedure call into a table.

The values that the 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.

References

Related statements: SELECT, DECLARE, DESCRIBE, EXECUTE, FLUSH, OPEN, PREPARE, CREATE EXTERNAL TABLE, and PUT

For a task-oriented discussion of inserting data into tables, see the [Informix Guide to SQL: Tutorial](#).

For a discussion of the GLS aspects of the INSERT statement, see the [Informix Guide to GLS Functionality](#).

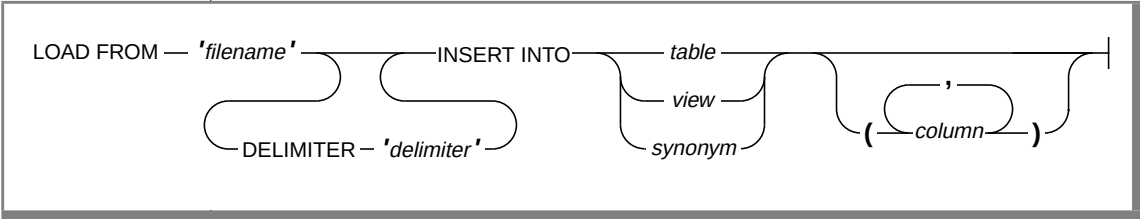
- +
- DB
- SQL

LOAD

Use the LOAD statement to insert data from an operating-system file into an existing table, synonym, or view.

Use this statement with DB-Access and SQL Editor.

Syntax



Element	Purpose	Restrictions	Syntax
column	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. 4-113
delimiter	Quoted string that identifies the character to use to separate the data values in each line of the load file The default delimiter is the character set in the DBDELIMITER environment variable. If DBDELIMITER has not been set, the default delimiter is the vertical bar ().	You cannot use any of the following characters as a delimiter character: backslash (\), newline character (=CTRL-J), and hexadecimal numbers (0-9, a-f, A-F).	Quoted String, p. 4-157

Element	Purpose	Restrictions	Syntax
<i>filename</i>	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 <i>column</i> parameter, the fields in the load file must match the columns specified for the table in number, order, and type. You must 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 BYTE and TEXT data types. For information on these restrictions, see “LOAD FROM File” on page 2-385 .	Quoted String, p. 4-157 . Pathname and filename must conform to the naming conventions of your operating system.
<i>synonym</i>	Name of the synonym to which you want to insert data	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table to which you want to insert data	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view to which you want to insert data	The view must exist.	Database Object Name, p. 4-25

(2 of 2)

Usage

The LOAD statement appends new rows to the table. It does not overwrite existing data.

You cannot add a row that has the same key as an existing row.

To use the LOAD statement, you must have Insert privileges for the table where you want to insert data. For information on database-level and table-level privileges, see the GRANT statement on [page 2-342](#).

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 the database server expects you to represent the data types in the LOAD file (when you 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.
Date	Character string in the following format: <i>mm/dd/year</i> 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.
MONEY	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 different currency notation if you indicate this notation with the DBMONEY environment variable.

(1 of 2)

Type of Data	Input Format
NULL	Nothing between the delimiters
Time	Character string in the following format: <i>year-month-day hour:minute:second.fraction</i> 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.

(2 of 2)

For more information on **DB** environment variables, refer to the [Informix Guide to SQL: Reference](#). For more information on **GL** environment variables, refer to the [Informix Guide to GLS Functionality](#).

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 [Informix Guide to GLS Functionality](#). ♦

GLS

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
- Newline character anywhere in the value of a VARCHAR or NVARCHAR column
- Newline character at end of a value for a TEXT value

Do not use the backslash character (\) as a field separator. It serves as an escape character to inform the LOAD statement that the next character is to be interpreted as part of the data.

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 files that contain BYTE or TEXT or VARCHAR data types, note the following information:

- If you give the LOAD statement data in which the character fields (including VARCHAR) are longer than the column size, the excess characters are disregarded.
- You cannot have leading and trailing blanks in BYTE fields.
- Use the backslash (\) to escape embedded delimiter and backslash characters in all character fields, including VARCHAR and TEXT.
- Data being loaded into a BYTE column must be in ASCII-hexadecimal form. BYTE columns cannot contain preceding blanks.
- Do not use the following as delimiting characters in the LOAD FROM file: 0 to 9, a to f, A to F, backslash, newline character.

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
```

DELIMITER Clause

Use the **DELIMITER** clause to specify the delimiter that separates the data contained in each column in a row in the input file. You can specify **TAB** (CTRL-I) or **<blank>** (= ASCII 32) as the delimiter symbol. You cannot use the following items as the delimiter symbol:

- Backslash (\)
- Newline character (= CTRL-J)
- Hexadecimal numbers (0 to 9, a to f, A to F)

If you omit this clause, the database server checks the **DBDELIMITER** environment variable. For information about how to set the **DBDELIMITER** environment variable, see the [Informix Guide to SQL: Reference](#).

The following example identifies the semicolon (;) as the delimiting character. The example uses Windows NT file-naming conventions.

```
LOAD FROM 'C:\data\loadfile' DELIMITER ';'
INSERT INTO orders
```

INSERT INTO Clause

Use the INSERT INTO clause to specify the table, synonym, or view in which to load the new data. You must specify the column names only if one of the following conditions is true:

- You are not loading data into all columns.
- The input file does not match the default order of the columns (determined when the table was created).

The following example identifies the **price** and **discount** columns as the only columns in which to add data. The example uses Windows NT file-naming conventions.

```
LOAD FROM 'C:\tmp\prices' DELIMITER ','  
INSERT INTO norman.worktab(price,discount)
```

References

Related statements: UNLOAD and INSERT

For a task-oriented discussion of the LOAD statement and other utilities for moving data, see the [Informix Migration Guide](#).

For a discussion of the GLS aspects of the LOAD statement, see the [Informix Guide to GLS Functionality](#).

+

LOCK TABLE

Use the LOCK TABLE statement to control access to a table by other processes.

Syntax

LOCK TABLE *table* IN *SHARE* MODE |
synonym *EXCLUSIVE*

Element	Purpose	Restrictions	Syntax
<i>synonym</i>	Name of the synonym for the table to be locked	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table to be locked	The table cannot be locked in exclusive mode by another process before you execute the LOCK TABLE statement. The table cannot be locked in share mode by another process before you execute a LOCK TABLE EXCLUSIVE statement.	Database Object Name, p. 4-25

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 database server configuration.
- All row and table locks release automatically after a transaction is completed. Note that 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

Related statements: BEGIN WORK, COMMIT WORK, ROLLBACK WORK, SET ISOLATION, SET LOCK MODE, and UNLOCK TABLE

For a discussion of concurrency and locks, see the [Informix Guide to SQL: Tutorial](#).

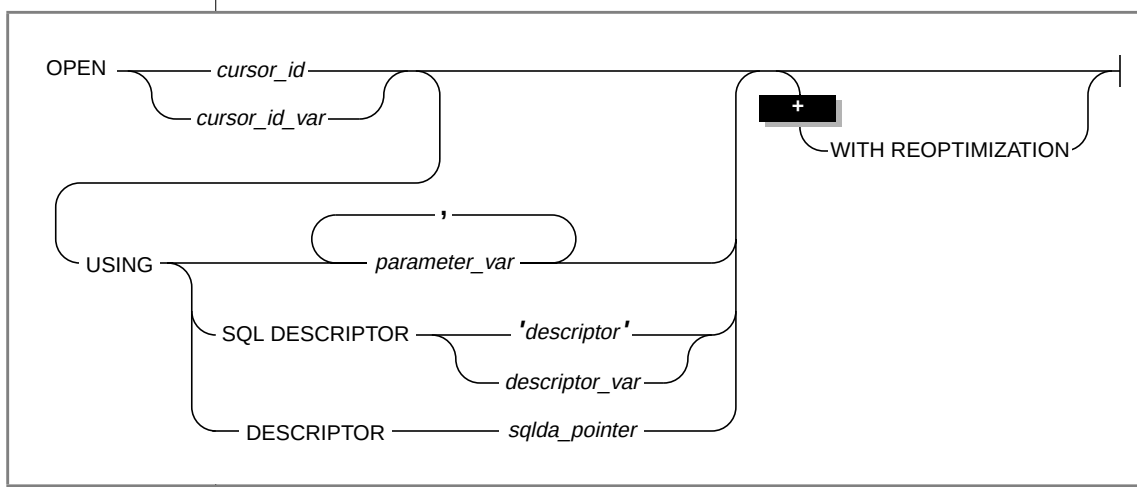
E/C

OPEN

Use the OPEN statement to activate a cursor associated with an EXECUTE PROCEDURE, INSERT, or SELECT statement, and thereby begin execution of the EXECUTE PROCEDURE, INSERT, or SELECT statement.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of a cursor	Cursor must have been previously created by a DECLARE statement.	Identifier, p. 4-113
<i>cursor_id_var</i>	Host variable that holds the value of <i>cursor_id</i>	Host variable must be a character data type. Cursor must have been previously created by a DECLARE statement.	Name must conform to language-specific rules for variable names
<i>descriptor</i>	Quoted string that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 4-157

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>descriptor_var</i>	Host variable that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 4-157
<i>parameter_var</i>	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Name must conform to language-specific rules for variable names.
<i>sqlda_pointer</i>	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 <i>sqlda_pointer</i> with a dollar sign (\$) or a colon (:). You must use an sqlda structure if you are using dynamic SQL statements.	DESCRIBE, p. 2-262

(2 of 2)

Usage

Use the OPEN statement to open a declared cursor and pass the associated EXECUTE PROCEDURE, INSERT, or SELECT statement to the database server to begin execution. When the program has retrieved or inserted all necessary rows, close the cursor with the CLOSE statement.

The specific actions that the database server takes differ, depending on the statement with which the cursor is associated.

When you associate an EXECUTE PROCEDURE, INSERT, or SELECT statement with a cursor directly (that is, you do not prepare the statement and associate the statement identifier with the cursor) the OPEN statement implicitly prepares the statement.

In an ANSI-compliant database, you receive an error code if you try to open a cursor that is already open. ♦

Opening a Select Cursor

When you open either a select cursor or an update cursor that is created with the SELECT... FOR UPDATE syntax, the SELECT statement is passed to the database server along with any values that are specified in the USING clause. (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.

ANSI

Example of Opening a Select Cursor

The following example illustrates a simple OPEN statement in ESQL/C:

```
EXEC SQL declare s_curs cursor for
        select * from orders;
EXEC SQL open s_curs;
```

Opening an Update Cursor Inside a Transaction

If you are working in a database with explicit transactions, you must open an update cursor within a transaction. This requirement is waived if you declared the cursor using the WITH HOLD keyword.

Opening a Procedure Cursor

When you open a procedure cursor, the EXECUTE PROCEDURE statement is passed to the database server along with any values that are specified in the USING Clause. The values are passed as arguments to the stored procedure, and the procedure must be declared to accept values. (If the statement was previously prepared, the statement passed to the database server when it was prepared.) The database server executes the procedure to the point of the first set of values returned by the procedure.

The following example illustrates a simple OPEN statement in ESQL/C:

```
EXEC SQL declare s_curs cursor for
        execute procedure new_proc();
EXEC SQL open s_curs;
```

Errors Associated with Select and Procedure Cursors

Because the database server is seeing the query for the first time, many errors are detected. The database server does not actually return the first row of data, but it sets a return code in the **sqlca.sqlcode**, **SQLCODE** field of the **sqlca**. The return code value is either negative or zero, as the following table describes.

Return Code Value	Meaning
Negative	Shows an error is detected in the SELECT statement
Zero	Shows the SELECT statement is valid

If the SELECT, SELECT...FOR UPDATE, or EXECUTE PROCEDURE statement is valid, but no rows match its criteria, the first FETCH statement returns a value of 100 (SQLNOTFOUND), which means no rows were found.



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

Opening an Insert Cursor

When you open an insert cursor, the cursor passes the INSERT statement to the database server, which checks the validity of the keywords and column names. The database server also allocates memory for an insert buffer to hold new data. (See the DECLARE statement on [page 2-241](#).)

An OPEN statement for a cursor that is associated with an INSERT statement cannot include a USING clause.

The following ESQL/C example illustrates an OPEN statement with an insert cursor:

```
EXEC SQL prepare s1 from
      'insert into manufact values ('npr', 'napier')';
EXEC SQL declare in_curs cursor for s1;
EXEC SQL open in_curs;
EXEC SQL put in_curs;
EXEC SQL close in_curs;
```

Reopening a Select or Procedure Cursor

The values that are named in the USING clause are evaluated only when the cursor is opened. While the cursor is open, subsequent changes to program variables in the USING clause do not change the active set of selected rows. The active set remains constant until a subsequent OPEN statement closes the cursor and reopens it or until the program closes the open cursor, which releases the active set.

Reopening the cursor creates a new active set that is based on the current values of the variables. 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 rows in the active set can be different in the following instances:

- for a select cursor, if data in the table was modified since the previous OPEN statement.
- for a procedure cursor, if the procedure takes a different execution path from the previous OPEN statement.

Reopening an Insert Cursor

When you reopen an insert cursor that is already open, you effectively flush the insert buffer; any rows that are stored in the insert buffer are written into the database table. The database server first closes the cursor, which causes the flush and then reopens the cursor. For information about how to check errors and count inserted rows, see the discussion of the PUT statement on [page 2-417](#).

USING Clause

The USING clause is required when the cursor is associated with a prepared SELECT statement that includes question-mark (?) placeholders. (See the PREPARE statement on [page 2-403](#).) You can supply values for these parameters in one of two ways. You can specify host variables in the USING clause, or you can specify a system-descriptor area in the USING SQL 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 statement question-mark (?) parameters in a one-to-one correspondence, from left to right.

You cannot include indicator variables in the list of variable names. To use an indicator variable, you must include the SELECT statement as part of the DECLARE statement.

The following example illustrates the USING clause with the OPEN statement in an 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_total;
```

USING SQL DESCRIPTOR Clause

You can also associate input values from a system-descriptor area. The keywords USING SQL DESCRIPTOR indicate the use of a system descriptor. This allows you to associate input values from a system-descriptor area and open a cursor.

If a system-descriptor area is used, the COUNT value specifies the number of input values that are described in occurrences of **sqlvar**. This number must correspond to the number of dynamic parameters in the prepared statement. The value of the COUNT field must be less than or equal to number of item descriptors that were specified when the system-descriptor area was allocated.

The following example shows the OPEN...USING SQL DESCRIPTOR clause:

```
EXEC SQL open selcurs using sql descriptor 'desc1';
```

USING DESCRIPTOR Clause

You can pass parameters for a prepared statement in the form of an **sqllda** pointer structure, which lists the data type and memory location of one or more values to replace question-mark (?) placeholders. For further information, refer to the **sqllda** discussion in the [INFORMIX-ESQL/C Programmer's Manual](#). The following example shows the OPEN...USING DESCRIPTOR clause in ESQL/C:

```
struct sqllda *sdp;
.
.
EXEC SQL open selcurs using descriptor sdp;
```

WITH REOPTIMIZATION Option

The WITH REOPTIMIZATION option allows you to reoptimize your query-design plan. When you prepare a SELECT statement or an EXECUTE PROCEDURE statement, your database 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 PROCEDURE 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 PROCEDURE 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:

```
EXEC SQL open selcurs using descriptor sdp with reoptimization;
```

Relationship Between OPEN and FREE

The database server allocates resources to prepared statements and open cursors. If you execute a `FREE statement_id` or `FREE statement_id_var` statement, you can still open the cursor associated with the freed statement ID. However, if you release resources with a `FREE cursor_id` or `FREE cursor_id_var` statement, you cannot use the cursor unless you declare the cursor again.

Similarly, if you use the `SET AUTOFREE` statement for one or more cursors, when the program closes the specific cursor, database server automatically frees the cursor-related resources. In this case, you cannot use the cursor or unless you declare the cursor again.

References

Related statements: `ALLOCATE DESCRIPTOR`, `DEALLOCATE DESCRIPTOR`, `DESCRIBE`, `CLOSE`, `DECLARE`, `EXECUTE`, `FETCH`, `FLUSH`, `FREE`, `GET DESCRIPTOR`, `PREPARE`, `PUT`, `SET AUTOFREE`, `SET DEFERRED_PREPARE`, and `SET DESCRIPTOR`

For a task-oriented discussion of the `OPEN` statement, see the [Informix Guide to SQL: Tutorial](#).

For more information on system-descriptor areas and the `sqlda` structure, refer to the [INFORMIX-ESQL/C Programmer's Manual](#).

+

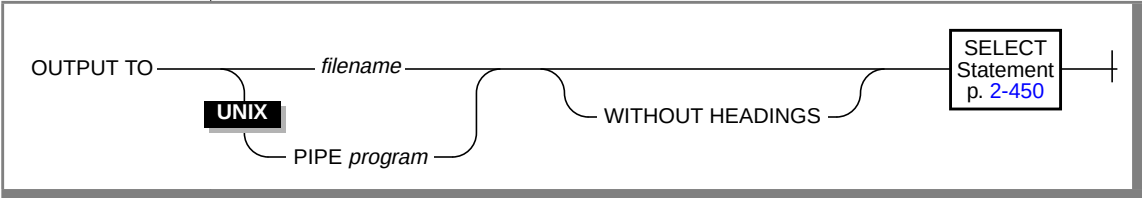
DB

OUTPUT

Use the OUTPUT statement to send query results directly to an operating-system file or to pipe query results to another program.

Use this statement with DB-Access.

Syntax



Element	Purpose	Restrictions	Syntax
<i>filename</i>	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 <i>filename</i> . 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.
<i>program</i>	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 use the OUTPUT statement to direct the results of a query to an operating-system file or to a program. You can also specify whether column headings should be omitted from the query output.

Sending Query Results to a File

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.

The following examples show how to send the result of a query to an operating-system file. The example uses UNIX file-naming conventions.

```
OUTPUT TO /usr/april/query1
SELECT * FROM cust_calls WHERE call_code = 'L'
```

Displaying Query Results Without Column Headings

You can display the results of a query without column headings by using the `WITHOUT HEADINGS` keywords.

UNIX

Sending Query Results to Another Program

In the UNIX environment, you can use the keyword `PIPE` to send the query results to another program, as the following example shows:

```
OUTPUT TO PIPE more
SELECT customer_num, call_dtime, call_code
FROM cust_calls
```

References

Related Statements: `SELECT` and `UNLOAD`

+

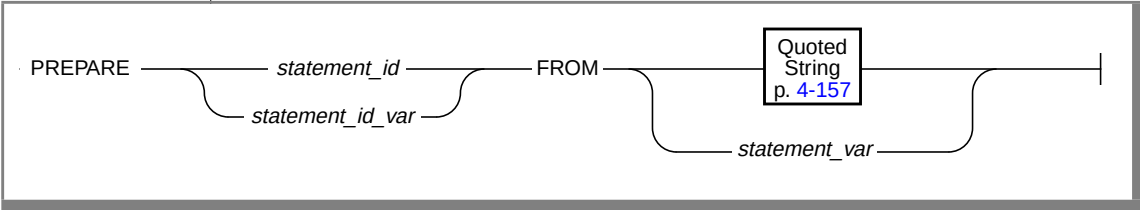
E/C

PREPARE

Use the PREPARE statement to parse, validate, and generate an execution plan for SQL statements at runtime.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>statement_id</i>	Identifier that represents the data structure of an SQL statement or sequence of SQL statements	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. 4-113
<i>statement_id_var</i>	Host variable that contains the statement identifier	This variable must be a character data type.	Name must conform to language-specific rules for variable names.
<i>statement_var</i>	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 2-408 and “Restrictions for Multistatement Prepares” on page 2-415.	Name must conform to language-specific rules for variable names.

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 items in a single program is limited by the available memory. These items include both statement identifiers that are named in PREPARE statements (*statement_id* or *statement_id_var*) and cursor declarations that incorporate EXECUTE PROCEDURE, INSERT, or SELECT 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 it contains no syntax errors, the text converts 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.

Scope of Statement Identifiers

A program can consist of one or more source-code files. By default, the scope of a statement identifier is global to the program. Therefore, a statement identifier that is prepared in one file can be referenced from another file.

In a multiple-file program, if you want to limit the scope of a statement identifier to the file in which it is prepared, preprocess all the files with the **-local** command-line option.

Releasing a Statement Identifier

A statement identifier can represent only one SQL statement or 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.

The first example shows a statement identifier that was prepared as a 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-identifier name 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 1-6](#).
- The text can contain either a single SQL statement or a sequence of statements that are separated by semicolons. For more information on how to prepare a single SQL statement, see [“SQL Statements Permitted in Single-Statement Prepares” on page 2-408](#). For more information on how to prepare a sequence of SQL statements, see [“Preparing Sequences of Multiple SQL Statements” on page 2-413](#).
- 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 PROCEDURE 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 more information on how to use identifiers in statement text, see [“Preparing Statements with SQL Identifiers” on page 2-410](#).
- Use a question mark (?) as a placeholder to indicate where data is supplied when the statement executes. For more information on how to use question marks as placeholders, see [“Preparing Statements That Receive Parameters” on page 2-409](#).
- 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 ESQL/C:

```
EXEC SQL prepare new_cust from  
      'insert into customer(fname,lname) values(?,?)';
```

Executing Stored Procedures Within a PREPARE Statement

You can prepare an EXECUTE PROCEDURE statement as long as it does not contain an INTO clause. The way to execute a prepared stored procedure depends on whether the stored procedure returns values:

- If the stored procedure does not return values (the procedure does not contain the RETURN statement), use the EXECUTE statement to execute the EXECUTE PROCEDURE statement.
- If the stored procedure returns only one row, you can use the INTO clause of the EXECUTE statement to specify host variables to hold the return values. Using EXECUTE...INTO to execute a stored procedure that returns more than one row generates a runtime error.
- If the stored procedure returns more than one row, you must associate the prepared EXECUTE PROCEDURE statement with a cursor using the DECLARE statement. You execute the statement with the OPEN statement and retrieve return values into host variables with the INTO clause of the FETCH statement.

If you do not know the number and data types of the values that a stored procedure returns, you must use a dynamic management structure to hold the returned values. You can use a system-descriptor area and manage it with SQL statements such as ALLOCATE DESCRIPTOR and GET DESCRIPTOR or you can use an **sqllda** structure. However, a system-descriptor area conforms to the X/Open standards.

SQL Statements Permitted in Single-Statement Prepares

In general, you can prepare any database-manipulation statement. You can prepare any single SQL statement except the following statements.

ALLOCATE DESCRIPTOR	GET DIAGNOSTICS
CLOSE	INFO
CONNECT	LOAD
DEALLOCATE DESCRIPTOR	OPEN
DECLARE	OUTPUT
DESCRIBE	PREPARE
DISCONNECT	PUT
EXECUTE	SET AUTOFREE
EXECUTE IMMEDIATE	SET CONNECTION
FETCH	SET DEFERRED_PREPARE
FLUSH	SET DESCRIPTOR
FREE	UNLOAD
GET DESCRIPTOR	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 clause. This clause is normally used with the DECLARE statement to create an update cursor. The following example shows a SELECT statement with a FOR UPDATE clause in ESQL/C:

```
sprintf(up_query, "%s %s %s",
        "select * from customer ",
        "where customer_num between ? and ? ",
        "for update");
EXEC SQL prepare up_sel from :up_query;

EXEC SQL declare up_curs cursor for up_sel;

EXEC SQL open up_curs using :low_cust,:high_cust;
```

Preparing Statements When Parameters Are Known

In some prepared statements, all necessary information is known at the time the statement is prepared. The following example in ESQL/C shows two statements that were prepared from constant data:

```
sprintf(redo_st, "%s %s",
        "drop table workt1; ",
        "create table workt1 (wtk serial, wtv float)" );
EXEC SQL prepare redotab from :redo_st;
```

Preparing Statements That Receive Parameters

In some statements, parameters are unknown when the statement is prepared because a different value can be inserted each time the statement is executed. In these statements, you can use a question-mark (?) placeholder where a parameter must be supplied when the statement is executed.

The PREPARE statements in the following ESQL/C examples show some uses of question-mark (?) placeholders:

```
EXEC SQL prepare s3 from
    'select * from customer where state matches ?';

EXEC SQL prepare in1 from
    'insert into manufact values (?, ?, ?)';

sprintf(up_query, "%s %s",
        "update customer set zipcode = ?"
        "where current of zip_cursor");
EXEC SQL prepare update2 from :up_query;
```

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 2-410](#).

The following example of an ESQL/C code fragment prepares a statement from a variable that is named **demoquery**. The text in the variable includes one question-mark (?) placeholder. The prepared statement is associated with a cursor and, when the cursor is opened, the USING clause of the OPEN statement supplies a value for the placeholder.

```
EXEC SQL BEGIN DECLARE SECTION;
    char queryvalue [6];
    char demoquery  [80];
EXEC SQL END DECLARE SECTION;

EXEC SQL connect to '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;
strcpy("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.

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 question-mark (?) placeholder for the database name in the DATABASE statement.
- You can use the question-mark (?) placeholder for the dbspace name in the IN *dbspace* clause of the CREATE DATABASE statement.
- You can use the question-mark (?) 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 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 **sqllda** descriptor and fetches each row with the descriptor. The fetch puts each row into memory locations that the program provides dynamically.

If a program retrieves all the rows in the active set, the FETCH statement would be placed in a loop that fetched each row. If the FETCH statement retrieves more than one data value (column), another loop exists after the FETCH, which performs some action on each data value.

```
#include <stdio.h>
EXEC SQL include sqllda;
EXEC SQL include sqltypes;

char *malloc( );

main()
{
    struct sqllda *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->sqlld; i++ )
        prsqllda (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;
            demodesc->sqlvar[i].sqlllen++;
            demodesc->sqlvar[i].sqldata =
                malloc( demodesc->sqlvar[i].sqlllen );
            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->sqlid; i++ )
    free( demodesc->sqlvar[i].sqldata );

printf ("Program Over.\n");
}

prsqlida(sp)
{
    struct sqlvar_struct *sp;
    {
        printf ("type = %d\n", sp->sqltype);
        printf ("len = %d\n", sp->sqlllen);
        printf ("data = %lx\n", sp->sqldata);
        printf ("ind = %lx\n", sp->sqlind);
        printf ("name = %s\n", sp->sqlname);
    }
}

```

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 the **sqlca.sqlerrd[4]** field in the **sqlca** to find the offset of the errors.

In a multistatement prepare, if no rows are returned from a WHERE clause in the following statements, you get SQLNOTFOUND (100):

- UPDATE...WHERE...
- SELECT INTO TEMP...WHERE...
- INSERT INTO...WHERE...
- DELETE FROM...WHERE...

In the following example, four SQL statements are prepared into a single 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 as exceptions in [“SQL Statements Permitted in Single-Statement Prepares” on page 2-408](#), you cannot use the following statements in text that contains multiple statements that are separated by semicolons.

CLOSE DATABASE	DROP DATABASE
CREATE DATABASE DATABASE	SELECT (except SELECT INTO TEMP)

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, 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 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

Related statements: DECLARE, DESCRIBE, EXECUTE, FREE, INFO, SET AUTOFREE, and SET DEFERRED_PREPARE

For information about basic concepts relating to the PREPARE statement, see the [Informix Guide to SQL: Tutorial](#).

For information about more advanced concepts relating to the PREPARE statement, see the [INFORMIX-ESQL/C Programmer's Manual](#).

+

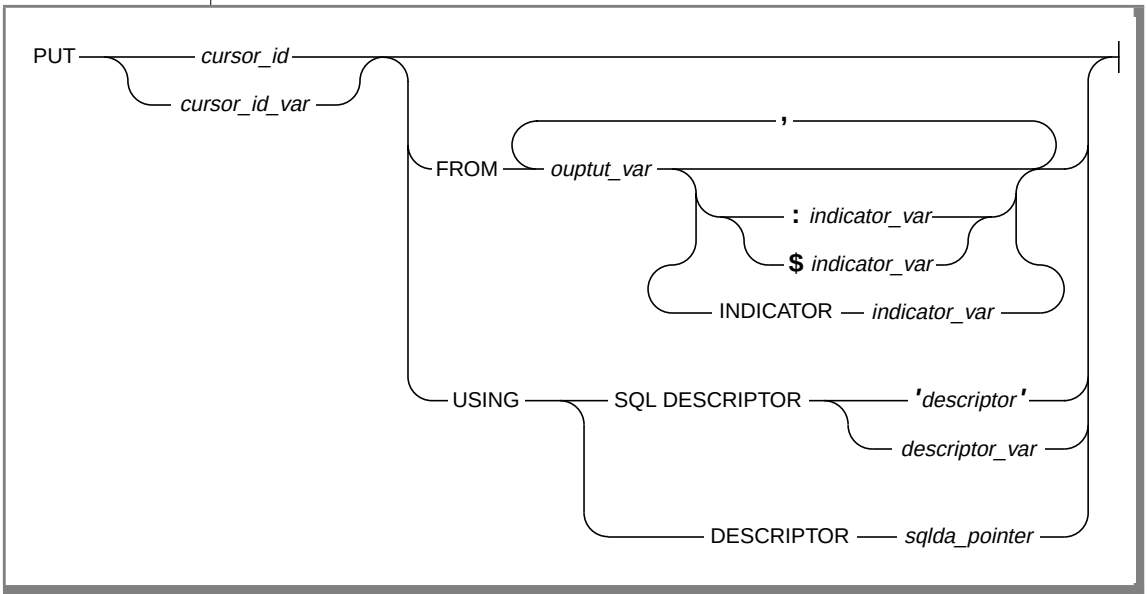
E/C

PUT

Use the PUT statement to store a row in an insert buffer for later insertion into the database.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of a cursor	A DECLARE statement must have previously created the cursor.	Identifier, p. 4-113
<i>cursor_id_var</i>	Host variable that holds the value of <i>cursor_id</i>	Host variable must be a character data type. A DECLARE statement must have previously created the cursor.	Name must conform to language-specific rules for variable names.
<i>descriptor</i>	Quoted string that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 4-157

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>descriptor_var</i>	Host variable name that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 4-157
<i>indicator_var</i>	Host variable that receives a return code if null data is placed in the corresponding data variable	Variable cannot be a DATETIME or INTERVAL data type.	Name must conform to language-specific rules for variable names.
<i>output_var</i>	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Name must conform to language-specific rules for variable names.
<i>sqlda_pointer</i>	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 (:).	DESCRIBE, p. 2-262.

(2 of 2)

Usage

Each PUT statement stores a row in an insert buffer that was created when the cursor 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.

If the current database uses explicit transactions, you must execute a PUT statement within a transaction.

The following example uses a PUT statement in ESQL/C:

```
EXEC SQL prepare ins_mcode from
'insert into manufact values(?,?)';
EXEC SQL declare mcode cursor for ins_mcode;
EXEC SQL open mcode;
EXEC SQL put mcode from :the_code, :the_name;
```

The PUT statement is not an X/Open SQL statement. Therefore, you get a warning message if you compile a PUT statement in X/Open mode. ♦

X/O

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 in memory addressed 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 ESQL/C example, 99 empty customer records are inserted into the **customer** table. Because all values are constants, no disk output occurs until the cursor closes. (The constant zero for **customer_num** causes generation of a SERIAL value.)

```
int count;
EXEC SQL declare fill_c cursor for
    insert into customer(customer_num) values(0);
EXEC SQL open fill_c;
for (count = 1; count <= 99; ++count)
    EXEC SQL put fill_c;
EXEC SQL close fill_c;
```

Naming Program Variables in INSERT

When the INSERT statement is written as part of the cursor declaration (in the DECLARE statement), you can name program variables in the VALUES clause. When each PUT statement is executed, the contents of the program variables at that time are used to populate the row that is inserted into the buffer.

If you are creating an insert cursor (using DECLARE with INSERT), you must use only program variables in the VALUES clause. Variable names are not recognized in the context of a prepared statement; you associate a prepared statement with a cursor through its statement identifier.

The following ESQL/C example illustrates the use of an insert cursor. The code includes the following statements:

- The DECLARE statement associates a cursor called **ins_curs** with an INSERT statement that inserts data into the **customer** table. The VALUES clause 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;
while ( (sqlca.sqlcode == 0) && (keep_going) )
{
    keep_going = get_user_input(cust_rec); /* ask user for new customer */
    if (keep_going) /* user did supply customer info */
    {
        cust_rec.customer_num = 0; /* request new serial value */
        EXEC SQL put ins_curs;
    }
    if (sqlca.sqlcode == 0) /* no error from PUT */
        keep_going = (prompt_for_y_or_n("another new customer") == 'Y')
    }
EXEC SQL close ins_curs;
```

Naming Program Variables in PUT

When the INSERT statement is prepared (see THE PREPARE statement on [page 2-403](#)), 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 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

You can create a system-descriptor area that describes the data type and memory location of one or more values. You can then specify that system-descriptor area in the USING SQL DESCRIPTOR clause of the PUT statement.

The following ESQL/C example shows how to associate values from a system-descriptor area:

```
EXEC SQL put selcurs using sql descriptor 'desc1';
```

Using an sqlda Structure

You can create an **sqlda** structure that describes the data type and memory location of one or more values. Then you can specify the **sqlda** structure in the USING DESCRIPTOR clause of the PUT statement. 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 usage of the PUT...USING DESCRIPTOR clause:

```
EXEC SQL put selcurs using descriptor pointer2;
```

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 BYTE or TEXT data (flushed after a single PUT statement).

If the program terminates without closing an insert cursor, the buffer remains unflushed. Rows that were inserted into the buffer since the last flush are lost. Do not rely on the end of the program to close the cursor and flush the buffer.

Error Checking

The **sqlca** contains information on the success of each PUT statement as well as information that lets you count the rows that were inserted. The result of each PUT statement is contained in the following fields of the **sqlca**: `sqlca.sqlcode`, `SQLCODE` and `sqlca.sqlerrd[2]`.

Data buffering with an insert cursor means that errors are not discovered until the buffer is flushed. For example, an input value that is incompatible with the data type of the column for which it is intended is discovered only when the buffer is flushed. When an error is discovered, rows in the buffer that are located after the error are *not* inserted; they are lost from memory.

The `SQLCODE` field is set to 0 if no error occurs; otherwise, it is set to an error code. The third element of the **sqlerrd** array is set to the number of rows that 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 also exists. For information about how to get the message text, check the **GET DIAGNOSTICS** statement.

Counting Total and Pending Rows

To count the number of rows actually inserted in the database and the number not yet inserted, perform the following steps:

1. Prepare two integer variables (for example, **total** and **pending**).
2. When the cursor is opened, set both variables to 0.
3. Each time a PUT statement executes, increment both **total** and **pending**.
4. Whenever a PUT or FLUSH statement executes, or the cursor closes, subtract the third field of the **SQLERRD** array from **pending**.

At any time, (**total** - **pending**) represents the number of rows 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

Related statements: CLOSE, FLUSH, DECLARE, and OPEN

For a task-oriented discussion of the PUT statement, see the [Informix Guide to SQL: Tutorial](#).

+

IDS

RENAME COLUMN

Use the RENAME COLUMN statement to change the name of a column.
You can use this statement only with Dynamic Server.

Syntax

RENAME COLUMN *table* *old_column* TO *new_column*

Element	Purpose	Restrictions	Syntax
<i>new_column</i>	New name to be assigned to the column	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. For more information on this restriction, see “How Triggers Are Affected” on page 2-426 .	Identifier, p. 4-113
<i>old_column</i>	Current name of the column you want to rename	The column must exist within the table.	Identifier, p. 4-113
<i>table</i>	Name of the table in which the column exists	The table must exist.	Database Object Name, p. 4-25

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.

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

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

+

RENAME DATABASE

Use the RENAME DATABASE statement to change the name of a database.

Syntax

RENAME DATABASE *old_database* TO *new_database*

Element	Purpose	Restrictions	Syntax
<i>new_database</i>	New name for 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. 4-22
<i>old_database</i>	Current name of the database	The database must exist.	Database Name, p. 4-22

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

Related statement: CREATE DATABASE

+

RENAME TABLE

Use the RENAME TABLE statement to change the name of a table.

Syntax

RENAME TABLE *old_table* TO *new_table*

Element	Purpose	Restrictions	Syntax
<i>new_table</i>	New name for the table	You cannot use the <i>owner.</i> convention in the new name of the table.	Identifier, p. 4-113
<i>old_table</i>	Current name of the table	The table must exist.	Identifier, p. 4-113

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.

AD/XP

If you are using Dynamic Server with AD and XP Options, you cannot rename a table that contains a dependent GK index. ♦

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. For further information on the **sysviews** system catalog table, see the [Informix Guide to SQL: Reference](#).

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

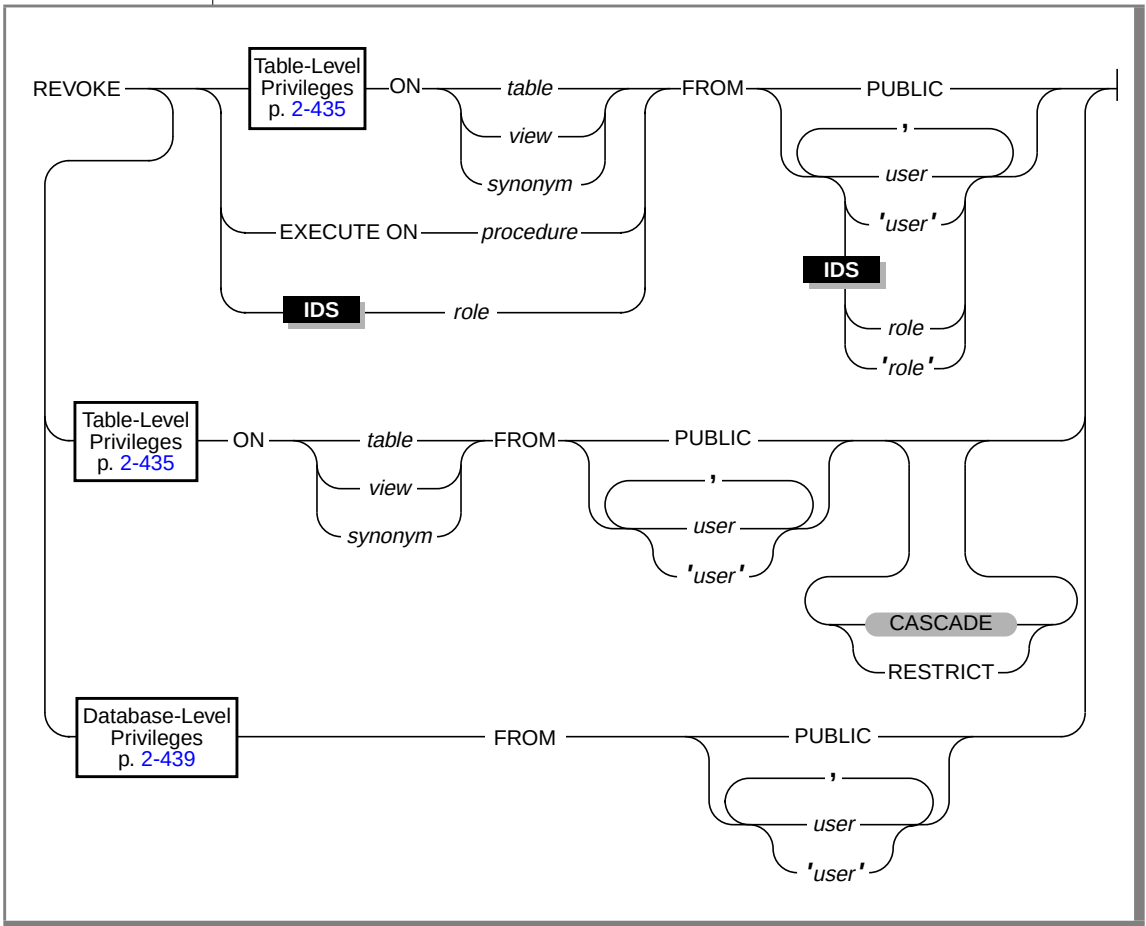
Related statements: ALTER TABLE, CREATE TABLE, DROP TABLE, and RENAME COLUMN

+

REVOKE

Use the REVOKE statement to revoke privileges on a database, table, or view, or on a procedure or a role.

Syntax



REVOKE

Element	Purpose	Restriction	Syntax
<i>procedure</i>	Name of the procedure for which the EXECUTE privilege is revoked	The name must be an existing procedure name.	Database Object Name, p. 4-25
<i>role</i>	Role from which a privilege or another role is to be revoked or role to be revoked from a user or another role	The role must have been created with the CREATE ROLE statement and granted with the GRANT statement	Identifier, p. 4-113
<i>synonym</i>	Synonym name for which a privilege is revoked	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Table name for which a privilege is revoked	The name must be an existing table name.	Database Object Name, p. 4-25
<i>user</i>	User from whom a privilege or role is revoked	The user must be a valid user.	Identifier, p. 4-113
<i>view</i>	View name for which a privilege is revoked	The name must be an existing view name.	Database Object Name, p. 4-25

Usage

You can use the REVOKE statement for the following purposes:

- You can revoke privileges on a table or view from a user or a role.
- You can revoke the privilege to execute a procedure from a user or a role.
- You can revoke privileges on a database from a user.
- You can revoke a role from a user or from another role.

You can use the REVOKE statement with the GRANT statement to control finely the ability of users to modify the database as well as to access and modify data in the tables.

If you use the PUBLIC keyword after the FROM keyword, the REVOKE statement revokes privileges from all users.

You can revoke all or some of the privileges that you granted to other users. No one can revoke privileges that another user grants.

ANSI

If you revoke the EXECUTE privilege on a stored procedure from a user, that user can no longer run that procedure using either the EXECUTE PROCEDURE or CALL statements.

If you use quotes, *user* appears exactly as typed.

In an ANSI-compliant database, if you do not enclose *user* in quotation marks, the name of the user is stored in uppercase letters. ♦

Users cannot revoke privileges from themselves.

Using the REVOKE Statement with Roles

You can use the REVOKE statement to remove privileges from a role and remove a role from a user or another role. Once a role is revoked from a user, the user cannot enable that role. You can revoke all or some of the roles granted to a user or role.

If a role is revoked from a user, the privileges associated with the role cannot be acquired by the user with the SET ROLE statement. However, this does not affect the currently acquired privileges.

You can use the REVOKE statement to revoke table-level privileges from a role; however, you cannot use the RESTRICT or CASCADE clauses when you do so.

Only the DBA or a user granted a role with the WITH GRANT OPTION can revoke privileges for a role.

If you revoke the Execute privilege on a stored procedure from a role, that role can no longer run that procedure.

Users cannot revoke roles from themselves. When you revoke a role, you cannot revoke the WITH GRANT OPTION separately. If the role was granted with the WITH GRANT OPTION, both the role and grant option are revoked.

The following example revokes the **engineer** role from the user **maryf**:

```
REVOKE engineer FROM maryf
```


Revoking Privileges Granted from 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*. In this case, when you revoke privileges from *user*, you automatically revoke the privileges of all users who received privileges from *user* or from the chain that *user* created. You can also specify this default condition with the CASCADE keyword.

Controlling the Scope of a REVOKE with the RESTRICT Option

Use the RESTRICT keyword to control the success or failure of the REVOKE command based on the existence of dependencies on the database objects that are being revoked. The following list shows the dependencies that cause the REVOKE statement to fail when you use the RESTRICT keyword:

- The user from whom the privilege is to be revoked has granted this privilege to another user or users.
- A view depends on a Select privilege that is being revoked.
- A foreign-key constraint depends on a References privilege that is being revoked.

Failure of the REVOKE When the Revokee Has Granted a Privilege

A REVOKE statement with the RESTRICT keyword fails if the user from whom a privilege is being revoked has granted the same privilege to another user or users. However, the same REVOKE statement does not fail if the revokee has the right to grant the privilege to other users but has not actually granted the privilege to any other user. We can illustrate these points by means of examples.

Assume that the user **clara** has granted the Select privilege on the **customer** table to the user **ted**, and she has also granted user **ted** the right to grant the Select privilege to other users. User **ted** has used this authority to grant the Select privilege on the **customer** table to the user named **tanja**. Now user **clara** attempts to revoke the Select privilege from user **ted** with the following REVOKE statement:

```
REVOKE SELECT ON customer FROM ted RESTRICT
```

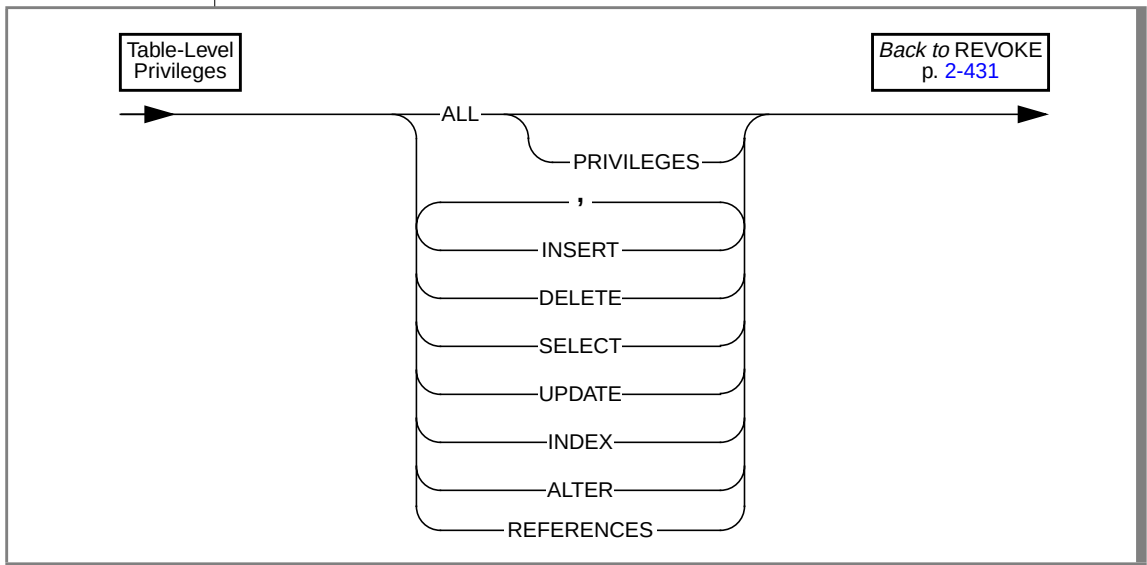
This statement fails because user **ted** has granted the Select privilege to user **tania**.

What if the revokee has the right to grant the privilege to other users but has not actually granted this privilege to any other user? For example, assume that the user **clara** has granted the Select privilege on the **customer** table to the user **roger**, and she has also granted user **roger** the right to grant the Select privilege to other users. However, user **roger** has not used this authority to granted the Select privilege to any other user. Now user **clara** attempts to revoke the Select privilege from user **roger** with the following REVOKE statement:

```
REVOKE SELECT ON customer FROM roger RESTRICT
```

This statement succeeds because user **roger** has not granted the Select privilege to any other user.

Table-Level Privileges



To revoke a table-level privilege from a user, you must revoke all occurrences of the privilege. For example, if two users grant the same privilege to a user, both of them must revoke the privilege. If one grantor revokes the privilege,

the user retains the privilege received from the other grantor. (The database server keeps a record of each table-level grant in the **syscolauth** and **systabauth** system catalog tables.)

If a table owner grants a privilege to PUBLIC, the owner cannot revoke the same privilege from any particular user. For example, if the table owner grants the Select privilege to PUBLIC and then attempts to revoke the Select privilege from **mary**, the REVOKE statement generates an error. The Select privilege was granted to PUBLIC, not to **mary**, and therefore the privilege cannot be revoked from **mary**. (ISAM error number 111, No record found, refers to the lack of a record in either the **syscolauth** or **systabauth** system catalog table, which would represent the grant that the table owner now wants to revoke.)

You can revoke table-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. Unlike the GRANT statement, the REVOKE statement does not allow you to qualify the Select, Update, or References privilege with a column name. Thus you cannot revoke access on specific columns.

Privilege	Function
INSERT	Provides the ability to insert rows
DELETE	Provides the ability to delete rows
SELECT	Provides the ability to display data obtained from a SELECT statement
UPDATE	Provides the ability to change column values
INDEX	Provides the ability to create permanent indexes You must have the Resource privilege to take advantage of the Index privilege. (Any user with the Connect privilege can create indexes on temporary tables.)

(1 of 2)

Privilege	Function
ALTER	Provides the ability to add or delete columns, modify column data types, and add or delete constraints This privilege also provides the ability to set the database object mode of unique indexes and constraints to the enabled, disabled, or filtering mode. In addition, this privilege provides the ability to set the database object mode of non-unique indexes and triggers to the enabled or disabled modes.
REFERENCES	Provides the ability to reference columns in referential constraints You must have the Resource privilege to take advantage of the References privilege. (However, you can add a referential constraint during an ALTER TABLE statement. This method does not require that you have the Resource privilege on the database.) Revoke the References privilege to disallow cascading deletes.
ALL	Provides all the preceding privileges The PRIVILEGES keyword is optional.

(2 of 2)

Behavior of the ALL Keyword

The ALL keyword revokes all table-level privileges. If any or all of the table-level privileges do not exist for the revokee, the REVOKE statement with the ALL keyword executes successfully but 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** wants to revoke all seven table-level privileges from user **hal**. So user **jocelyn** issues the following REVOKE statement:

```
REVOKE ALL ON customer FROM hal
```



This statement executes successfully but returns SQLSTATE code 01006. The SQLSTATE warning is returned with a successful statement as follows:

- The statement succeeds in revoking the Select and Insert privileges from user **hal** because user **hal** had those privileges.
- SQLSTATE code 01006 is returned because the other five privileges implied by the ALL keyword (the Delete, Update, References, Index, and Alter privileges) did not exist for user **hal**; therefore, these privileges were not revoked.

Tip: The ALL keyword instructs the database server to revoke everything possible, including nothing. If the user from whom privileges are revoked has no privileges on the table, the REVOKE ALL statement still succeeds because it revokes everything possible from the user (in this case, no privileges at all).

Examples of Revoking and Regranting Table-Level Privileges

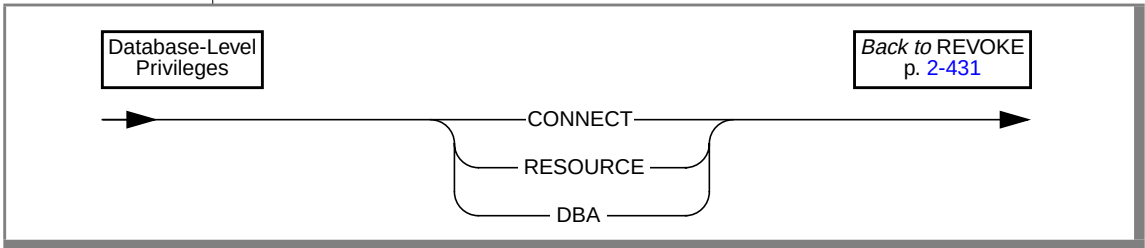
The following example revokes the Index and Alter privileges from all users for the **customer** table; these privileges are then granted specifically to user **mary**.

```
REVOKE INDEX, ALTER ON customer FROM PUBLIC;  
GRANT INDEX, ALTER ON customer TO mary;
```

Because you cannot revoke access on specific columns, when you revoke the Select, Update, or References privilege from a user, you revoke the privilege for all columns in the table. You must use a GRANT statement specifically to regrant any column-specific privilege that should be available to the user, as the following example shows:

```
REVOKE ALL ON customer FROM PUBLIC;  
GRANT ALL ON customer TO john, cathy;  
GRANT SELECT (fname, lname, company, city)  
ON customer TO PUBLIC;
```

Database-Level Privileges



Only a user with the DBA privilege can grant or revoke database-level privileges.

Three levels of database privileges control access. These privilege levels are, from lowest to highest, Connect, Resource, and DBA. To revoke a database privilege, specify one of the keywords CONNECT, RESOURCE, or DBA in the REVOKE statement.

Because of the hierarchical organization of the privileges (as outlined in the privilege definitions that are described later in this section), if you revoke either the Resource or the Connect privilege from a user with the DBA privilege, the statement has no effect. If you revoke the DBA privilege from a user who has the DBA privilege, the user retains the Connect privilege on the database. To deny database access to a user with the DBA or Resource privilege, you must first revoke the DBA or the Resource privilege and then revoke the Connect privilege in a separate REVOKE statement.

Similarly, if you revoke the Connect privilege from a user with the Resource privilege, the statement has no effect. If you revoke the Resource privilege from a user, the user retains the Connect privilege on the database.

The database privileges are associated with the following keywords.

Privilege	Functions
CONNECT	Lets you query and modify data. You can modify the database schema if you own the database object that you want to modify. Any user with the Connect privilege can perform the following functions: ■ Connect to the database with the CONNECT statement or another connection statement ■ Execute SELECT, INSERT, UPDATE, and DELETE statements, provided that the user has the necessary table-level privileges ■ Create views, provided that the user has the Select privilege on the underlying tables ■ Create synonyms ■ Create temporary tables, and create indexes on the temporary tables ■ Alter or drop a table or an index, provided that the user owns the table or index (or has the Alter, Index, or References privilege on the table) ■ Grant privileges on a table, provided that the user owns the table (or has been given privileges on the table with the WITH GRANT OPTION keyword)
RESOURCE	Lets you extend the structure of the database. In addition to the capabilities of the Connect privilege, the holder of the Resource privilege can perform the following functions: ■ Create new tables ■ Create new indexes ■ Create new procedures

(1 of 2)

Privilege	Functions
DBA	Lets the holder of DBA privilege perform the following functions in addition to the capabilities of the Resource privilege: ■ 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 tables ■ Drop any database object, regardless of who owns it ■ Create tables, views, and indexes as well as specify another user as owner of the database objects ■ Execute the DROP DATABASE statement ■ Insert, delete, or update rows of any system catalog table except systables

(2 of 2)



Warning: Although the user **informix** and DBAs can modify most system catalog tables (only the user **informix** can modify **systables**), Informix strongly recommends that you do not update, delete, or insert any rows in these tables. Modifying the system catalog tables can destroy the integrity of the database. Informix does support use of the ALTER TABLE statement to modify the size of the next extent of system catalog tables.

Effect of Uncommitted Transactions

When a REVOKE statement is executed, an exclusive row lock is placed on the entry in the **systables** system catalog table for the table from which privileges have been revoked. This lock is released only after the transaction that contains the REVOKE statement is complete. When another transaction attempts to prepare a SELECT statement against this table, the transaction fails because the entry for this table in **systables** is exclusively locked. The attempt to prepare the SELECT statement will not succeed until the first transaction has been committed.

References

Related statements: GRANT, GRANT FRAGMENT, and REVOKE FRAGMENT

For information about roles, see the following statements: CREATE ROLE, DROP ROLE, and SET ROLE.

For a discussion of database-level privileges and table-level privileges, see the *Informix Guide to Database Design and Implementation*.

For a discussion of how to embed GRANT and REVOKE statements in programs, see the [Informix Guide to SQL: Tutorial](#).

+

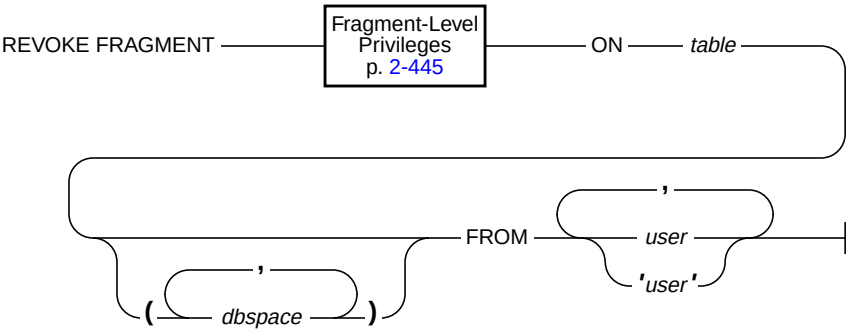
IDS

REVOKE FRAGMENT

Use the REVOKE FRAGMENT statement 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.

You can use this statement only with Dynamic Server. This statement is not available with Dynamic Server, Workgroup and Developer Editions.

Syntax



REVOKE FRAGMENT

Element	Purpose	Restrictions	Syntax
<i>dbspace</i>	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. 4-113
<i>table</i>	Name of the table that contains the fragment or fragments on which privileges are to be revoked No default value exists.	The specified table must exist and must be fragmented by expression.	Database Object Name, p. 4-25
<i>user</i>	Name of the user or users from whom the specified privileges are to be revoked No default value exists.	The user must be a valid user.	Identifier, p. 4-113

Usage

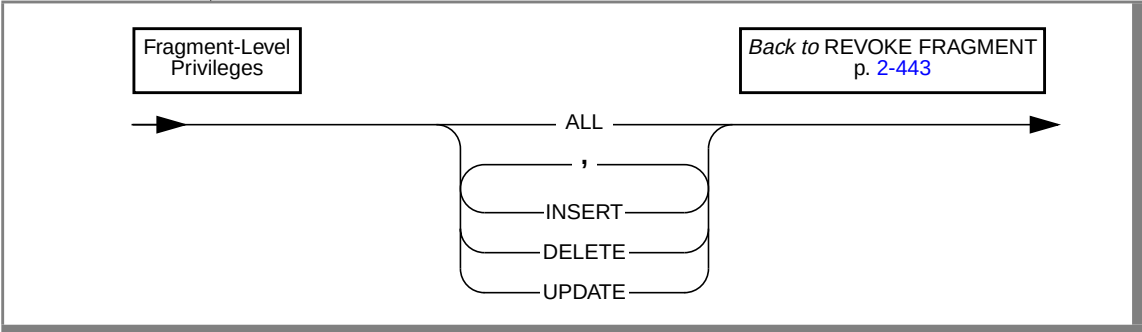
Use the REVOKE FRAGMENT statement to revoke the Insert, Update, or Delete privilege on one or more fragments of a fragmented table from one or more users.

The REVOKE FRAGMENT statement is only valid for tables that are fragmented according to an expression-based distribution scheme. For an explanation of expression-based distribution scheme, see the ALTER FRAGMENT statement on [page 2-8](#).

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 following table defines each of the fragment-level privileges.

Privilege	Function
ALL	Provides insert, delete, and update privileges on a fragment
INSERT	Provides the ability to insert rows in the fragment
DELETE	Provides the ability to delete rows in the fragment
UPDATE	Provides 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

Related statements: GRANT FRAGMENT and REVOKE

For a discussion of fragment-level and table-level privileges, see the *Informix Guide to Database Design and Implementation*.

ROLLBACK WORK

Use the ROLLBACK WORK statement to cancel a transaction deliberately and undo any changes that occurred since the beginning of the transaction. The ROLLBACK WORK statement restores the database to the state that it was in before the transaction began.

Syntax

```
ROLLBACK _____|
                WORK
```

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.

In an ANSI-compliant database, transactions are implicit. Transactions start after each COMMIT WORK or ROLLBACK WORK statement, so no BEGIN WORK statement is required. If you issue a ROLLBACK WORK statement when no transaction is pending, the statement is accepted but has no effect. ♦

In ESQL/C, the ROLLBACK WORK statement closes all open cursors except those that are declared with hold. Hold cursors remain open after a transaction is committed or rolled back.

ANSI**E/C**

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. This step prevents the program from looping if the ROLLBACK WORK statement encounters an error or a warning. ♦

WORK Keyword

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

```
ROLLBACK;
```

```
ROLLBACK WORK;
```

References

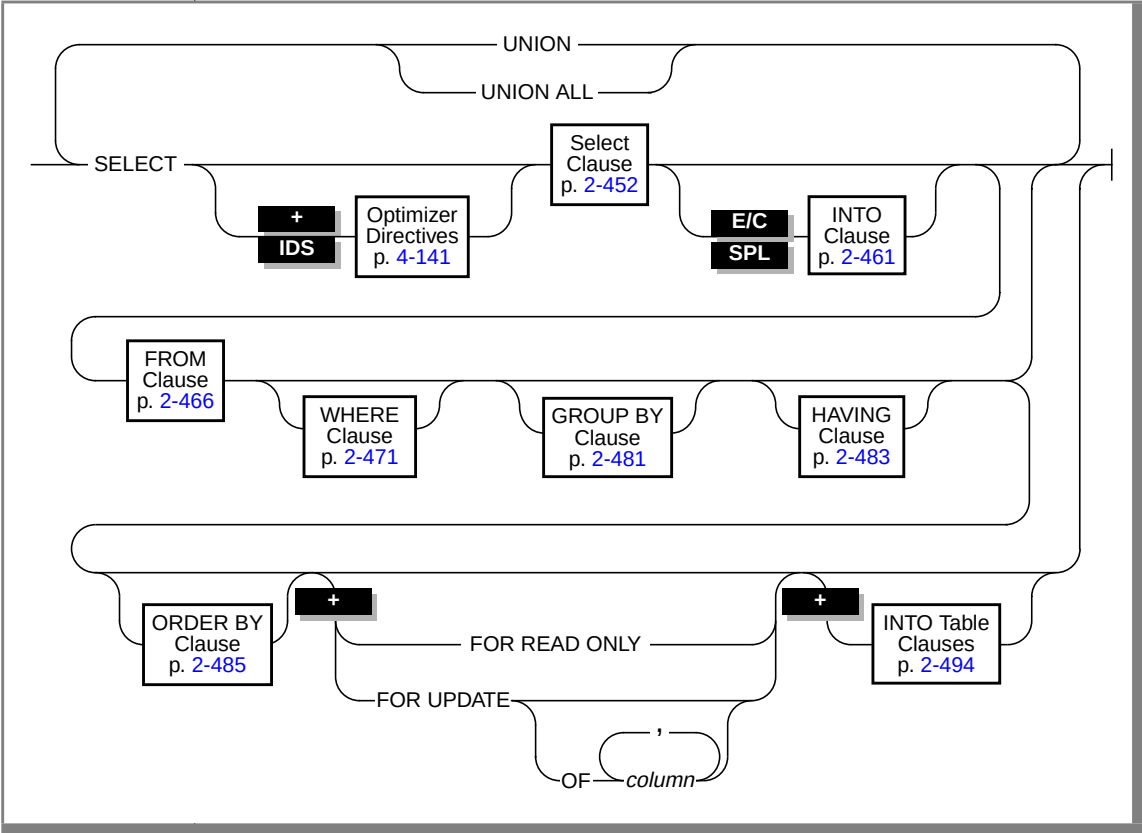
Related statements: BEGIN WORK and COMMIT WORK

For a discussion of transactions and ROLLBACK WORK, see the [Informix Guide to SQL: Tutorial](#).

SELECT

Use the SELECT statement to query a database.

Syntax



Element	Purpose	Restrictions	Syntax
<i>column</i>	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. 4-113

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.

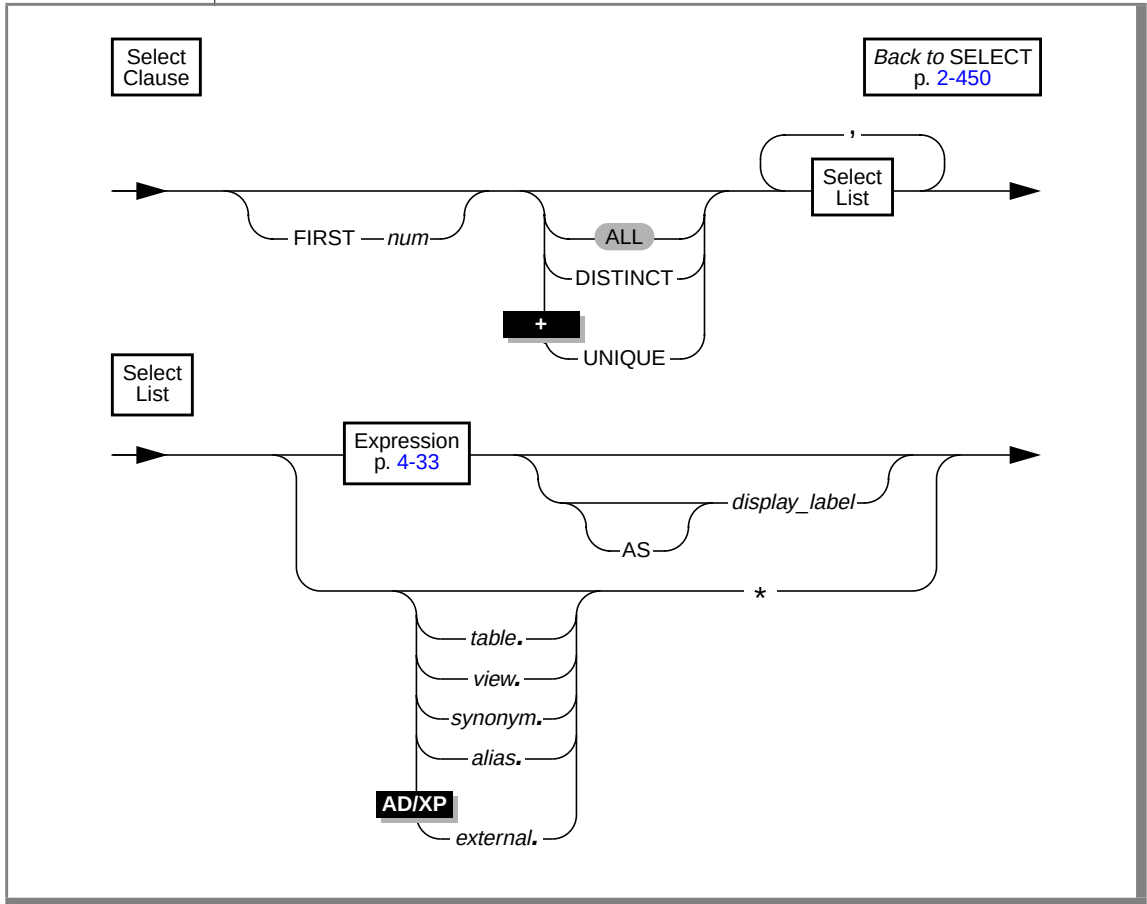
The SELECT statement includes many basic clauses. Each clause is described in the following list.

E/C, SPL

Clause	Purpose
SELECT	Names a list of items to be read from the database
INTO	Specifies the program variables or host variables that receive the selected data in ESQL / C or stored procedures
FROM	Names the tables that contain the selected columns
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
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
INTO TEMP	Creates a temporary table in the current database and puts the results of the query into the table
INTO SCRATCH	Creates an unlogging temporary table in the current database and puts the results of the query into the table
INTO EXTERNAL	Loads an external table with the results of the query

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
*	Symbol that 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.
<i>alias</i>	Temporary alternative name assigned to the table or view in the FROM clause For more information on aliases, see “FROM Clause” on page 2-466.	You cannot use an alias for a SELECT clause unless you assign the alias to the table or view in the FROM clause.	Identifier, p. 4-113
<i>display_label</i>	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/C, the value of <i>display_label</i> is stored in the sqlname field of the sqllda structure. For more information on the <i>display_label</i> parameter, see “Using a Display Label” on page 2-460.	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 or SELECT...INTO EXTERNAL 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. 4-113

(1 of 2)

SELECT

Element	Purpose	Restrictions	Syntax
<i>external</i>	Name of the external table from which you want to retrieve data	The external table must exist.	Database Object Name, p. 4-25
<i>num</i>	Integer that indicates the number of rows to return	The value must be greater than 0. If the value is greater than the number of rows that match the selection criteria of the query, all matching rows are returned.	Literal Number, p. 4-139
<i>synonym</i>	Name of the synonym from which you want to retrieve data	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table from which you want to retrieve data	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view from which you want to retrieve data	The view must exist.	Database Object Name, p. 4-25

(2 of 2)

In the SELECT clause, specify exactly what data is being selected as well as whether you want to omit duplicate values.

FIRST Clause

The FIRST clause allows you to specify a maximum number of rows to retrieve that match conditions specified in the SELECT statement. Rows that match the selection criteria, but fall outside the specified number, are not returned.

The following example retrieves at most 10 rows from a table:

```
SELECT FIRST 10 a, b FROM tab1;
```

When you use this option with an ORDER BY clause, you can retrieve the first number of rows according to the order criteria. For example, the following query finds the ten highest-paid employees.

```
SELECT FIRST 10 name, salary
FROM emp
ORDER BY salary DESC
```

AD/XP

If you are using Dynamic Server with AD and XP Options, you can also use the FIRST clause to select the first rows that result from a union query. In the following example, the FIRST clause is applied to the result of the UNION expression

```
SELECT FIRST 10 a, b FROM tab1 UNION SELECT a, b FROM tab2
```



Restrictions on the First Clause

The FIRST clause is not allowed in the following situations:

- When you define a view
- In nested SELECT statements
- In subqueries
- In the SELECT clause of an INSERT statement
- When your SELECT statement is selecting data and inserting it into another table, such as a temporary, scratch, or external table
- In stored procedures
- In statements that allow embedded SELECT statements to be used as expressions
- As part of a UNION query (if you are using Dynamic Server) ♦

IDS

IDS

Using FIRST as a Column Name with Dynamic Server

Although FIRST is a keyword, the database server can also interpret it as a column name. If an integer does not follow the keyword, the database server interprets FIRST as the name of a column. For example, if a table has columns **first**, **second**, and **third**, the following query would return data from the column named **first**:

```
SELECT first  
from T
```

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 stored procedure), or combination thereof, in the select list. The expression types are described in [“Expression” on page 4-33](#).

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 ESQL/C application or stored procedure variables in a stored procedure) 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. For a complete description of the syntax and use of column expressions, see [“Column Expressions” on page 4-35](#).

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. For a complete description of the syntax and use of constant expressions, see [“Constant Expressions” on page 4-47](#).

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 Stored Procedure Expressions

Stored procedures 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** procedure for each **customer_num** and displays the output of the procedure under the **n_orders** label:

```
SELECT customer_num, lname,
       get_orders(customer_num) n_orders
FROM customer
--
--get_orders is a stored procedure that
--finds the number of orders placed by a customer;
--(customer_num) is the argument to the procedure;
--n_orders is a display label for the return value.
```

If you know the name of the input parameter to a procedure, you can include this name in the stored-procedure expression, as the following example shows. The **SELECT** statement in this example is the same as in the preceding example, except that the stored-procedure expression includes the name of the input parameter and an equals (=) sign.

```
SELECT customer_num, lname,
       get_orders(custnum = customer_num) n_orders
FROM customer
--
--get_orders is a stored procedure that
--finds the number of orders placed by a customer;
--custnum is the name assigned
--to the input parameter of the stored procedure
--by a CREATE PROCEDURE statement;
--customer_num is the value assigned to custnum
--by the SELECT statement;
--n_orders is a display label for the return value.
```

When you use a stored-procedure expression in the select list, the stored procedure must be one that returns only a single value. If the procedure is one that returns multiple values, the **SELECT** statement fails and returns an error message. This message indicates that the procedure returns too many values.

For the complete syntax of stored-procedure expressions, see [“Procedure Call Expressions” on page 4-111](#).

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
```

Using a Display Label

If you are creating 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.

In DB-Access, a display label appears as the heading for that column in the output of the SELECT statement. ♦

In ESQL/C, the value of *display_label* is stored in the **sqlname** field of the **sqllda** structure. For more information on the **sqllda** structure, see the [INFORMIX-ESQL/C Programmer's Manual](#). ♦

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:

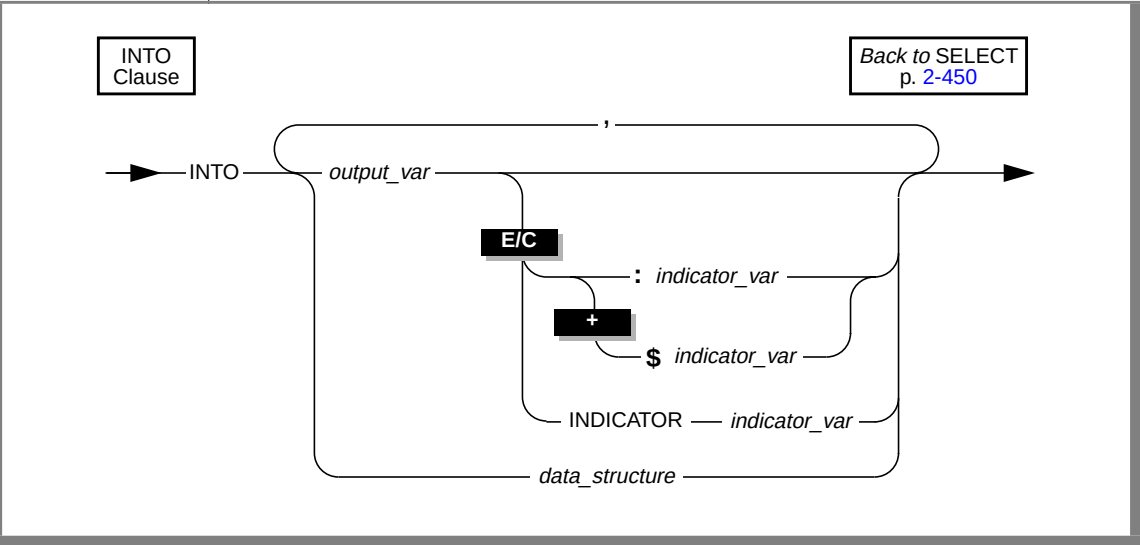
```
SELECT call_dtime AS minute FROM cust_calls
```

DB

E/C

INTO Clause

Use the INTO clause within a stored procedure or ESQL/C 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
<i>data_structure</i>	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.	Name must conform to language-specific rules for data structures.
<i>indicator_var</i>	Program variable that receives a return code if null data is placed in the corresponding <i>output_var</i>	This parameter is optional, but you should use an indicator variable if the possibility exists that the value of the corresponding <i>output_var</i> is null.	Name must conform to language-specific rules for variable names.

SELECT

Element	Purpose	Restrictions	Syntax
<i>output_var</i>	Program variable or host variable This variable receives the value of the corresponding item in the select list of the SELECT clause.	The order of receiving variables in the INTO clause must match the order of the corresponding items in the select list of the SELECT clause. The number of receiving variables must be equal to the number of items in the select list. The data type of each receiving variable should agree with the data type of the corresponding column or expression in the select list. For the actions that the database server takes when the data type of the receiving variable does not match that of the selected item, see “Warnings in ESQL/C” on page 2-465 .	Name must conform to language-specific rules for variable names.

(2 of 2)

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 singleton SELECT statement in 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

In ESQL/C, if the possibility exists that data returned from the SELECT statement is null, use an indicator variable in the INTO clause. For more information about indicator variables, see the [INFORMIX-ESQL/C Programmer's Manual](#).

INTO Clause with Cursors

If the SELECT statement returns more than one row, you must use a cursor in a FETCH statement to fetch the rows individually. You can put the INTO clause in the FETCH statement rather than in the SELECT statement, but you cannot put it in both.

The following 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;
```

Preparing a SELECT...INTO Query

In ESQL/C, you cannot prepare a query that has an INTO clause. You can prepare the query without the INTO clause, declare a cursor for the prepared query, open the cursor, and then use the FETCH statement with an INTO clause to fetch the cursor into the program variable. Alternatively, you can declare a cursor for the query without first preparing the query and include the INTO clause in the query when you declare the cursor. Then open the cursor, and fetch the cursor without using the INTO clause of the FETCH statement.

Using Array Variables with the INTO Clause

In ESQL/C, if you use a DECLARE statement with a SELECT statement that contains an INTO clause, and the 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 ESQL/C code example declares a cursor for a SELECT...INTO statement using the variables **i** and **j** as subscripts for the array **a**. After you declare the cursor, the INTO clause of the SELECT statement is equivalent to INTO a[5], a[2].

```
i = 5
j = 2
EXEC SQL declare c cursor for
    select order_num, po_num into :a[i], :a[j] from orders
    where order_num =1005 and po_num =2865
```

You can also use program variables in the FETCH statement to specify an element of a program array in the INTO clause. With the FETCH statement, the program variables are evaluated at each fetch rather than when you declare the cursor.

Error Checking

If the data type of the receiving variable does not match that of the selected item, the data type of the selected item is converted, if possible. If the conversion is impossible, an error occurs, and a negative value is returned in the status variable, **sqlca.sqlcode**, **SQLCODE**. In this case, the value in the program variable is unpredictable.

ANSI

In an ANSI-compliant database, if the number of variables that are listed in the INTO clause differs from the number of items in the SELECT clause, you receive an error. ♦

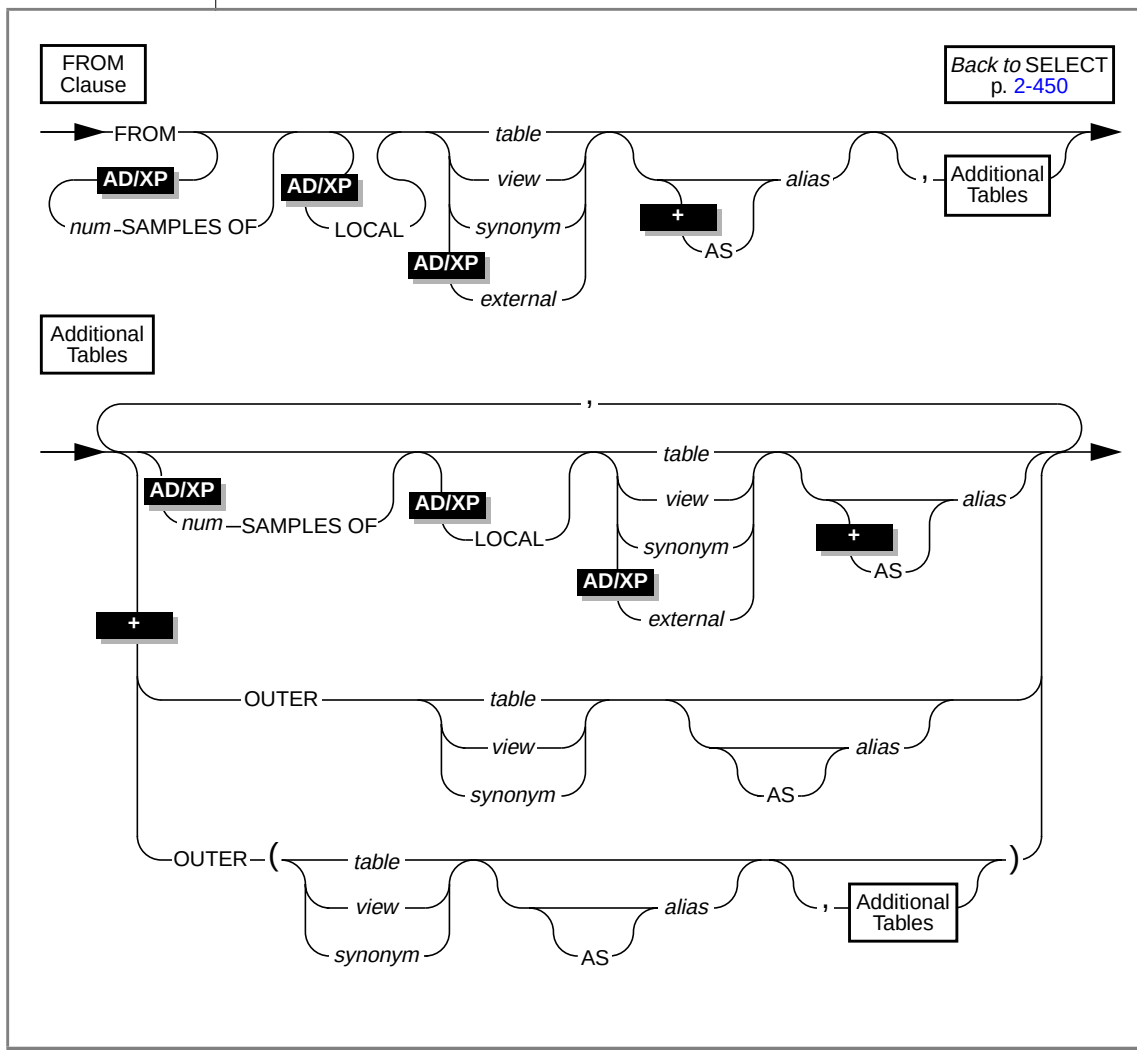
E/C

Warnings in ESQL/C

In ESQL/C, 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 **sqlwarn** structure: **sqlca.sqlwarn.sqlwarn3**. The actual number of variables that are transferred is the lesser of the two numbers. For information about the **sqlwarn** structure, see the [INFORMIX-ESQL/C Programmer's Manual](#).

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
<i>alias</i>	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. For further information on this restriction, see “AS Keyword with Aliases” on page 2-468 .	Identifier, p. 4-113
<i>external</i>	Name of the external table from which you want to retrieve data	The external table must exist.	Database Object Name, p. 4-25
<i>num</i>	Number of sample rows to return	The value must be an unsigned integer greater than 0. If the value specified is greater than the number of rows in the table, the whole table is scanned.	Literal Number, p. 4-139
<i>synonym</i>	Name of the synonym from which you want to retrieve data	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table from which you want to retrieve data	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view from which you want to retrieve data	The view must exist.	Database Object Name, p. 4-25

Use the keyword OUTER to form outer joins. Outer joins preserve rows that otherwise would be discarded by simple joins. For more information on outer joins, see the [Informix Guide to SQL: Tutorial](#).

You can supply an alias for a table name or view name in the FROM clause. If you do so, you must use the alias to refer to the table or view in other clauses of the SELECT statement. Aliases are especially useful with a self-join. For more information about self-joins, see the WHERE clause on [page 2-471](#).

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.

AD/XP

Restrictions on Using External Tables in Joins and Subqueries

In Dynamic Server with AD and XP Options, when you use external tables in joins or subqueries, the following restrictions apply:

- Only one external table is allowed in a query.
- The external table cannot be the outer table in an outer join.
- For subqueries that cannot be converted to joins, you can use an external table in the main query, but not the subquery.
- You cannot do a self join on an external table.

For more information on subqueries, refer to your [Performance Guide](#).

AD/XP

LOCAL Keyword

In Dynamic Server with AD and XP Options, the LOCAL table feature allows client applications to read data only from the *local* fragments of a table. In other words, it allows the application to read only the fragments that reside on the coserver to which the client is connected.

This feature provides application partitioning. An application can connect to multiple coservers, execute a LOCAL read on each coserver, and assemble the final result on the client machine.

You qualify the name of a table with the LOCAL keyword to indicate that you want to retrieve rows from fragments only on the local coserver. The LOCAL keyword has no effect on data that is retrieved from nonfragmented tables.

When a query involves a join, you must plan carefully if you want to extract data that the client can aggregate. The simplest way to ensure that a join will retrieve data suitable for aggregation is to limit the number of LOCAL tables to one. The client can then aggregate data with respect to that table.

The following example shows a query that returns data suitable for aggregation by the client:

```
SELECT x.col1, y.col2
FROM LOCAL tab1 x, tab2 y{can be aggregated by client}
INTO TEMP t1
WHERE x.col1 = y.col1 ;{tab1 is local}
```

The following example shows data that the client cannot aggregate:

```
SELECT x.col1, y.col2
FROM LOCAL tab1 x, LOCAL tab2 y{cannot be aggregated by
                                client}
INTO SCRATCH s4
WHERE x.col1 = y.col1 ;{tab1 and tab2 are local}
```

The client must submit exactly the same query to each coserver to retrieve data that can be aggregated.

AD/XP

Sampled Queries: the SAMPLES OF Option

In Dynamic Server with AD and XP Options, *sampled queries* are supported. Sampled queries are queries that are based on *sampled tables*. A sampled table is the result of randomly selecting a specified number of rows from the table, rather than all rows that match the selection criteria.

You can use a sampled query to gather quickly an approximate profile of data within a large table. If you use a sufficiently large sample size, you can examine trends in the data by sampling the data instead of scanning all the data. In such cases, sampled queries can provide better performance than scanning the data.

To indicate that a table is to be sampled, specify the number of samples to return in the SAMPLES OF option of the FROM clause within the SELECT statement. You can run sampled queries against tables and synonyms, but not against views. Sampled queries are not supported in the INSERT, DELETE, UPDATE, or other SQL statements.

A sampled query has at least one sampled table. You do not need to sample all tables in a sampled query. You can specify the SAMPLES OF option for some tables in the FROM clause but not specify it for other tables.

The sampling method is known as *sampling without replacement*. This term means that a sampled row is not sampled again. The database server applies selection criteria *after* samples are selected. Therefore, the database server uses the selection criteria to restrict the sample set, not the rows from which it takes the sample.



If a table is fragmented, the database server divides the specified number of samples among the fragments. The number of samples from a fragment is proportional to the ratio of the size of a fragment to the size of the table. In other words, the database server takes more samples from larger fragments.

Important: You must run `UPDATE STATISTICS LOW` before you run the query with the `SAMPLES OF` option. If you do not run `UPDATE STATISTICS`, the `SAMPLE` clause is ignored, and all data is returned. For better results, Informix recommends that you run `UPDATE STATISTICS MEDIUM` before you run the query with the `SAMPLES OF` option.

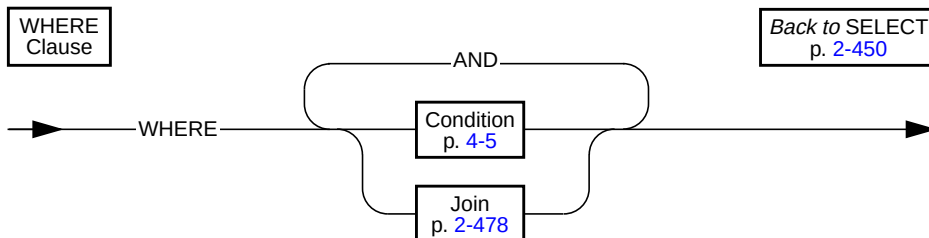
The results of a sampled query will contain a certain amount of deviation from a complete scan of all rows. However, you can reduce this expected error to an acceptable level by increasing the proportion of sampled rows to actual rows. When you use sampled queries in joins, the expected error increases dramatically; you must use larger samples in each table to retain an acceptable level of accuracy.

For example, you might want to generate a list of how many of each part is sold from the **parts_sold** table, which is known to contain approximately 100,000,000 rows. The following query provides a sampling ratio of 1 percent and returns an approximate result:

```
SELECT part_number, COUNT(*) * 100 AS how_many
FROM 1000000 SAMPLES OF parts_sold
GROUP BY part_number;
```

WHERE Clause

Use the `WHERE` clause to specify search criteria and join conditions on the data that you are selecting.



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; this 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 4-5](#).

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 originating 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 4-10](#).

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 > '6/04/98'

SELECT fname, lname, company
    FROM customer
    WHERE city[1,3] = 'San'
```

BETWEEN Condition

For a complete description of the BETWEEN condition, see [page 4-10](#).

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 '6/1/97' AND '9/1/97'

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 4-16](#).

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 4-11](#).

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 4-12](#).

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 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 4-11](#).

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 4-17](#).

With the EXISTS subquery, one or more columns can be returned.

The following example of a SELECT statement with an EXISTS subquery returns the stock number and manufacturer code for every item that has never been ordered (and is therefore not listed in the **items** table). It is appropriate to use an EXISTS subquery in this SELECT statement because you need the correlated subquery to test both **stock_num** and **manu_code** in the **items** table.

```
SELECT stock_num, manu_code FROM stock
WHERE NOT EXISTS
  (SELECT stock_num, manu_code FROM items
   WHERE stock.stock_num = items.stock_num AND
         stock.manu_code = items.manu_code)
```

The preceding example would work equally well if you use a SELECT * in the subquery in place of the column names because you are testing for the existence of a row or rows.

ALL/ANY/SOME Subquery

For a complete description of the ALL / ANY / SOME subquery, see [page 4-18](#).

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)
```

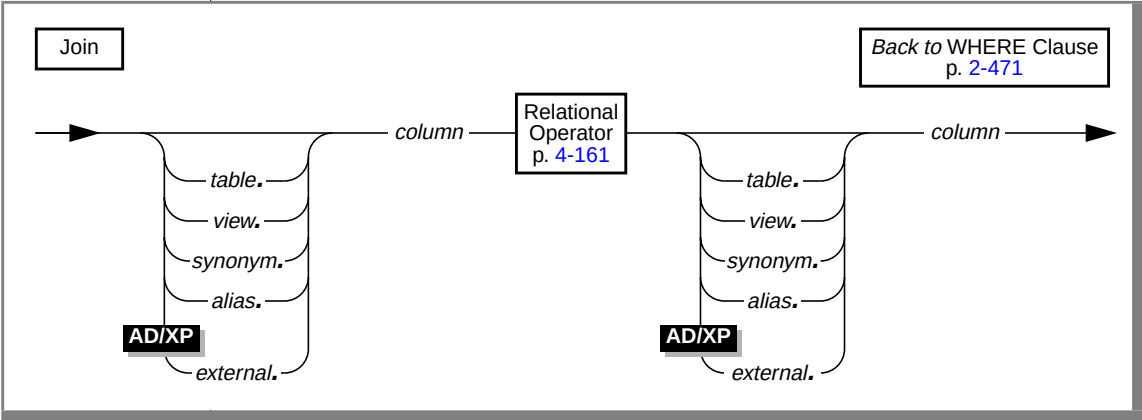
You can omit the keywords ANY, ALL, or SOME in a subquery if you know that the subquery returns exactly one value. If you omit ANY, ALL, or SOME, and the subquery returns more than one value, you receive an error. The subquery in the following example returns only one row because it uses an aggregate 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	Temporary alternative name assigned to the table or view in the FROM clause For more information on aliases for tables and views, see “FROM Clause” on page 2-466.	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. 4-113
column	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. 4-113

Element	Purpose	Restrictions	Syntax
<i>external</i>	Name of the external table from which you want to retrieve data	The external table must exist.	Database Object Name, p. 4-25
<i>synonym</i>	Name of the synonym to be joined	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table to be joined	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view to be joined	The view must exist.	Database Object Name, p. 4-25

(2 of 2)

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 specify the column where the two tables are joined in the *SELECT* list.

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*. For the full syntax of a table name, see “[Database Object Name](#)” on page 4-25.

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
```

AD/XP

If you are using Dynamic Server with AD and XP Options, you cannot use a self-join with an external table. ♦

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
```

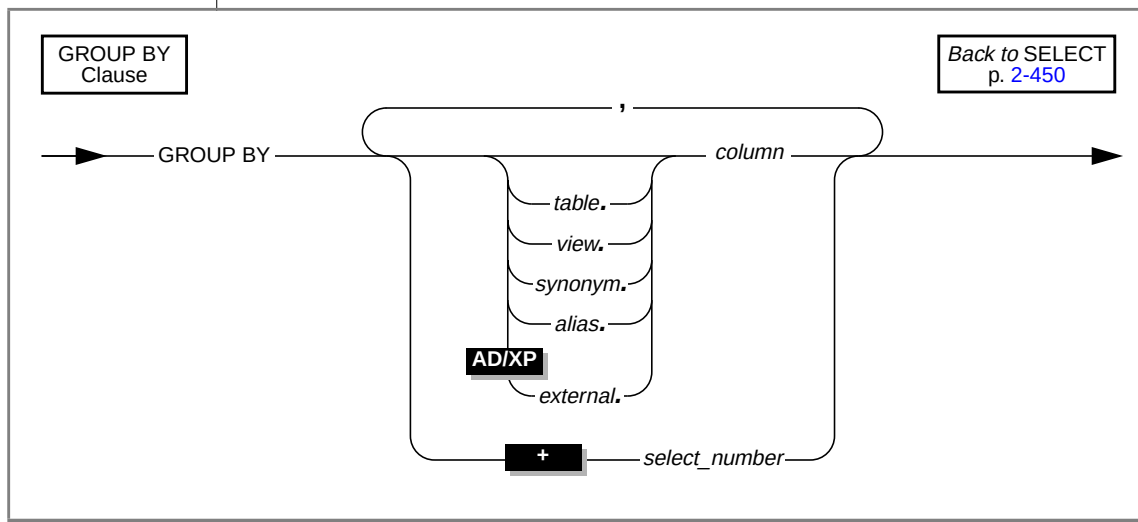
AD/XP

If you are using Dynamic Server with AD and XP Options, you cannot use an external table as the outer table in an outer join. ♦

For more information about outer joins, see the [Informix Guide to SQL: Tutorial](#).

GROUP BY Clause

Use the GROUP BY clause to produce a single row of results for each group. A group is a set of rows that have the same values for each column listed.



Element	Purpose	Restrictions	Syntax
<i>alias</i>	Temporary alternative name assigned to a table or view in the FROM clause For more information on aliases for tables and views, see “FROM Clause” on page 2-466 .	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. 4-113
<i>column</i>	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 <i>column</i> .	See “Relationship of the GROUP BY Clause to the SELECT Clause” on page 2-482 .	Identifier, p. 4-113
<i>external</i>	Name of the external table from which you want to retrieve data	The external table must exist.	Database Object Name, p. 4-25

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>select_number</i>	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 <i>select_number</i> .	See “Using Select Numbers” on page 2-483 .	Literal Number, p. 4-139
<i>synonym</i>	Name of the synonym where the column or columns exist	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table where the column or columns exist	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view where the column or columns exist	The view must exist.	Database Object Name, p. 4-25

(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.

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. Note that 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.

HAVING
Clause



HAVING

Condition
p. 4-5



Back to SELECT
p. 2-450

SELECT

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:

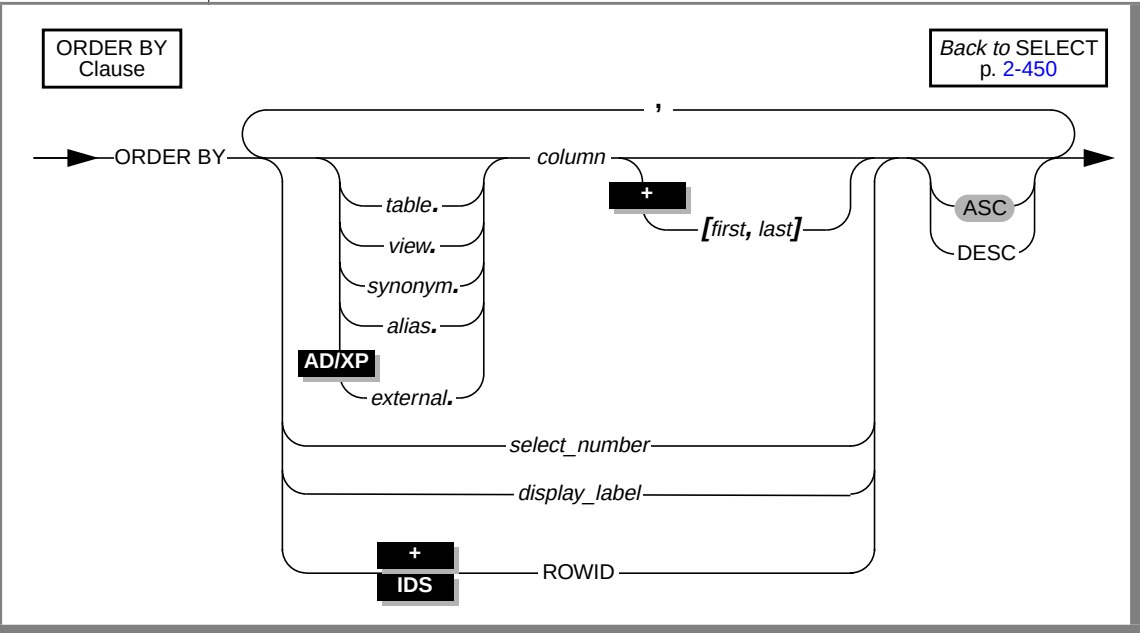
```
SELECT customer_num, EXTEND (call_dtime), call_code
      FROM cust_calls
      GROUP BY call_code, 2, 1
      HAVING customer_num < 120
```

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	Alias assigned to a table or view in the FROM clause. For more information on aliases for tables and views, see “FROM Clause” on page 2-466 .	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. 4-113

SELECT

Element	Purpose	Restrictions	Syntax
<i>column</i>	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.	Identifier, p. 4-113
<i>external</i>	Name of the external table from which you want to retrieve data	The external table must exist.	Database Object Name, p. 4-25
<i>display_label</i>	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. 4-113
<i>first</i>	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. 4-139
<i>last</i>	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. 4-139
<i>select_number</i>	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. 4-139

(2 of 3)

Element	Purpose	Restrictions	Syntax
<i>synonym</i>	Name of the synonym that contains the specified column	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table that contains the specified column	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view that contains the specified column	The view must exist.	Database Object Name, p. 4-25

(3 of 3)

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 [Informix 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. For a listing of the collating sequence, see [“Collating Order for English Data” on page 4-162](#).

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

If you are using Dynamic Server, 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 you are selecting from is a fragmented table that does not have a rowid column.

You cannot specify the **rowid** column in the ORDER BY clause unless you have included the **rowid** column in the select list of the SELECT clause.

For further information on how to use the **rowid** column in column expressions, see [“Expression” on page 4-33](#).

ORDER BY Clause with DECLARE

In ESQ/C, you cannot use a DECLARE statement with a FOR UPDATE clause to associate a cursor with a SELECT statement that has an ORDER BY clause.

Placing Indexes on ORDER BY Columns

When you include an ORDER BY clause in a SELECT statement, you can improve the performance of the query by creating an index on the column or columns that the ORDER BY clause specifies. The database server uses the index that you placed on the ORDER BY columns to sort the query results in the most efficient manner. For more information on how to create indexes that correspond to the columns of an ORDER BY clause, see [“ASC and DESC Keywords” on page 2-111](#) 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 (strcmp(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 = 'rosej' 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, INTO EXTERNAL, ORDER BY, UNION, or UNIQUE.

For information on how to declare an update cursor for a SELECT statement that does not include a FOR UPDATE clause, see [page 2-249](#).

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-compliant database, and you have not yet performed a level-0 backup of this data, the database is in read-only mode. When the database is in read-only mode, the database server rejects any attempts by a select cursor to access the data unless the SELECT or the DECLARE includes a FOR READ ONLY clause. This restriction remains in effect until the user has performed a level-0 backup of the data.

When the database is an ANSI-compliant database, select cursors are update cursors by default. An update cursor is a cursor that can be used to modify data. These update cursors are incompatible with the read-only mode of the database. For example, 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. ◆

IDS

If you are using Dynamic Server, see the [Guide to the High-Performance Loader](#) for more information on the express mode of the HPL. For more information on level-0 backups, see your [Backup and Restore Guide](#). For more information on select cursors, read-only cursors, and update cursors, see the DECLARE statement on [page 2-241](#). ◆

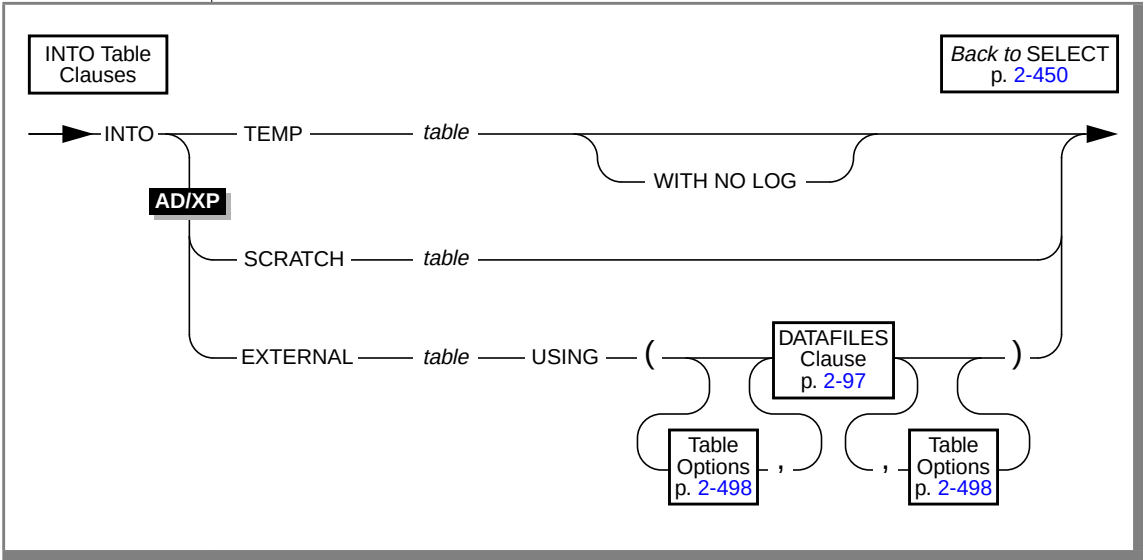
Syntax That Is Incompatible with the FOR READ ONLY Clause

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 2-251](#).

INTO Table Clauses

Use the INTO table clauses to specify a table to receive the data that the SELECT statement retrieves.



Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of a table that 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.	Database Object Name, p. 4-25

Naming Columns

The column names of the temporary, scratch, or external table are those that are 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, scratch or external table. If you do not provide a display label for a column expression, the table uses the column name from the select list.

The following INTO TEMP example creates the **pushdate** table with two columns, **customer_num** and **slowdate**:

```
SELECT customer_num, call_dtime + 5 UNITS DAY slowdate
FROM cust_calls INTO TEMP pushdate
```

Results When No Rows are Returned

When you use an INTO Table clause combined with the WHERE clause, and no rows are returned, the **SQLNOTFOUND** value is 100 in ANSI-compliant databases and 0 in databases that are not ANSI compliant. If the SELECT INTO TEMP...WHERE... statement is a part of a multistatement prepare and no rows are returned, the **SQLNOTFOUND** value is 100 for both ANSI-compliant databases and databases that are not ANSI-compliant.

E/C

Restrictions with INTO Table Clauses in ESQL/C

In ESQL/C, do not use the INTO clause with an INTO Table clause. If you do, no results are returned to the program variables and the **sqlca.sqlcode**, **SQLCODE** variable is set to a negative value.

Precedence of the INTO TEMP and INTO SCRATCH Clauses

If the **DBSPACETEMP** environment variable is set, temporary tables created with the INTO TEMP and INTO SCRATCH clauses 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 the **DBSPACETEMP** environment variable, see the [Informix Guide to SQL: Reference](#). For more information about the ONCONFIG parameter **DBSPACETEMP**, see your [Administrator's Guide](#).

INTO TEMP Clause

Use the INTO TEMP clause to create a temporary table that contains the query results. The initial and next extents for a temporary table are always eight pages.

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.

You can put indexes on a temporary table.

A logged, temporary table disappears when your program ends or when you issue a DROP TABLE statement on the temporary table.

If you are using Dynamic Server and your database does not have logging, the table behaves in the same way as a table that uses the WITH NO LOG Option. ♦

AD/XP

Using the WITH NO LOG Option

Use the WITH NO LOG option to reduce the overhead of transaction logging. If you use the WITH NO LOG option, operations on the temporary table are not included in the transaction-log operations.

The behavior of a temporary table that you create with the WITH NO LOG option is the same as that of a scratch table.

INTO SCRATCH Clause

If you are using Dynamic Server with AD and XP Options, use the INTO SCRATCH clause to reduce the overhead of transaction logging. (Operations on scratch tables are not included in transaction-log operations.)

A scratch table does not support indexes, constraints, or rollback.

A scratch table disappears when the first of the following three situations occurs:

- The program ends.
- A DROP TABLE statement is issued on the temporary table.
- The database is closed.

A scratch table is identical to a temporary table that is created with the WITH NO LOG option.

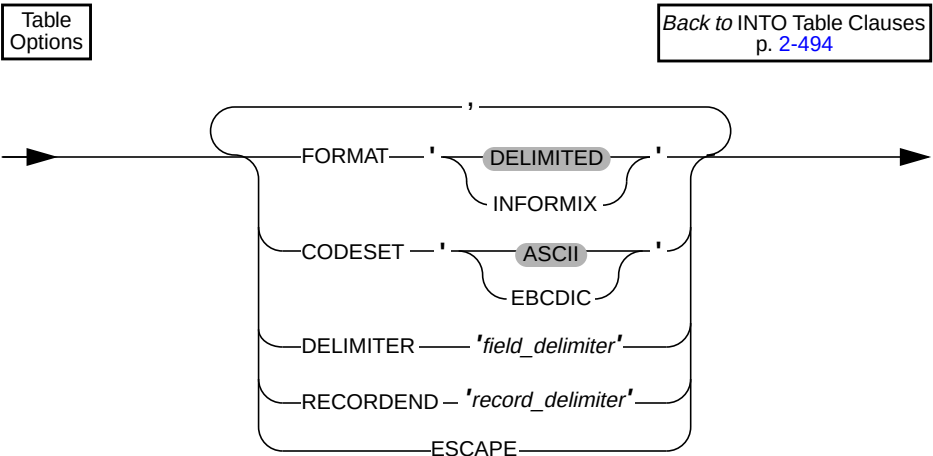
AD/XP

INTO EXTERNAL Clause

If you are using Dynamic Server with AD and XP Options, use the INTO EXTERNAL clause to build a SELECT statement that unloads data from your database into an external table. When you use the INTO EXTERNAL clause to unload data, you create a default external table description. This clause is especially useful for unloading Informix-internal data files because you can use the external table description when you subsequently reload the files.

To obtain the same effect for text tables, issue a CREATE EXTERNAL...SAMEAS statement. Then issue an INSERT INTO...SELECT statement.

Table Options



Element	Purpose	Restrictions	Syntax
<i>field_delimiter</i>	Character to separate fields The default value is the pipe () character.	If you use a non-printing character as a delimiter, you must encode it as the octal representation of the ASCII character. For example, '\006' can represent CTRL-F.	Quoted String, p. 4-157
<i>record_delimiter</i>	Character to separate records If you do not set the RECORDEND environment variable, the default value is the newline character (\n).	If you use a non-printing character as a delimiter, you must encode it as the octal representation of the ASCII character. For example, '\006' can represent CTRL-F.	Quoted String, p. 4-157

The following table describes the keywords that apply to unloading data. If you want to specify additional table options in the external-table description for the purpose of reloading the table later, see “Table Options” on page 2-99. In the SELECT...INTO EXTERNAL statement, you can specify all table options that are discussed in the CREATE EXTERNAL TABLE statement except the fixed-format option.

You can use the INTO EXTERNAL clause when the format type of the created data file is either DELIMITED text or text in Informix internal data format. You cannot use it for a fixed-format unload.

Keyword	Purpose
CODESET	Specifies the type of code set
DELIMITER	Specifies the character that separates fields in a delimited text file
ESCAPE	Directs the database server to recognize ASCII special characters embedded in ASCII-text-based data files If you do not specify ESCAPE when you load data, the database server does not check the character fields in text data files for embedded special characters. If you do not specify ESCAPE when you unload data, the database server does not create embedded hexadecimal characters in text fields.
FORMAT	Specifies the format of the data in the data files
RECORDEND	Specifies the character that separates records in a delimited text file

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.

IDS

E/C

Restrictions on a 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.
- The columns in the SELECT clause of each query cannot be BYTE or TEXT columns. This restriction does not apply to UNION ALL operations.
- If you use an ORDER BY clause, it must follow the last SELECT clause, and you must refer to the item ordered by integer, not by identifier. Ordering takes place after the set operation is complete.
- In Dynamic Server, you cannot use a UNION operator inside a subquery. ♦
- In ESQL/C, 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 1997 and the first quarter of 1998.

```
SELECT customer_num, call_code FROM cust_calls
    WHERE call_dtime BETWEEN
        DATETIME (1997-1-1) YEAR TO DAY
        AND DATETIME (1997-3-31) YEAR TO DAY

UNION ALL

SELECT customer_num, call_code FROM cust_calls
    WHERE call_dtime BETWEEN
        DATETIME (1998-1-1) YEAR TO DAY
        AND DATETIME (1998-3-31) YEAR TO DAY
```

If you want to remove duplicates, use the UNION operator without the keyword ALL in the query. In the preceding example, if the combination 101 B were returned in both SELECT statements, a UNION operator would cause the combination to be listed once. (If you want to remove duplicates within each SELECT statement, use the DISTINCT keyword in the SELECT clause, as described on [page 2-452](#).)

References

For task-oriented discussions of the SELECT statement, see the [Informix Guide to SQL: Tutorial](#).

For a discussion of the GLS aspects of the SELECT statement, see the [Informix Guide to GLS Functionality](#).

+

IDS

E/C

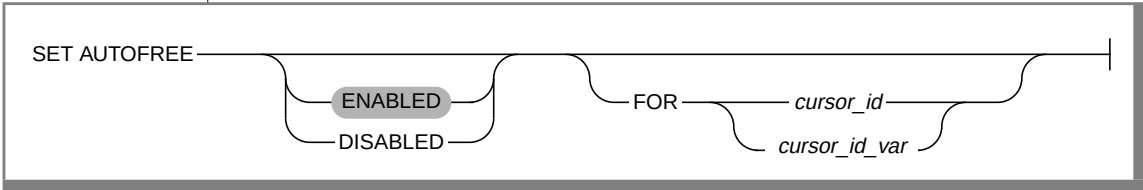
SET AUTOFREE

Use the SET AUTOFREE statement to specify that the database server free the memory allocated for a cursor automatically, as soon as the cursor is closed.

You can use this statement only with Dynamic Server.

Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of a cursor for which the Autofree feature is enabled or disabled	The cursor must be declared within the program.	Identifier, p. 4-113
<i>cursor_id_var</i>	Host variable that holds the value of <i>cursor_id</i>	The host variable must store the name of a cursor that is declared within the program.	Name must conform to language-specific rules for variable names.

Usage

When the Autofree feature is enabled for a cursor, you do not need to explicitly use a FREE statement to free the cursor memory in the database server once the cursor is closed.

You can specify the ENABLED or DISABLED options for the SET AUTOFREE statement. If you do not specify either option, the default is ENABLED. The following example shows how to enable the Autofree feature for all subsequent cursors in the program by default:

```
EXEC SQL set autofree;
```

Restrictions

The SET AUTOFREE statement that enables the Autofree feature must appear before the OPEN statement that opens a cursor. If a cursor is already open, the SET AUTOFREE statement does not affect its behavior.

After a cursor is autofree-enabled, you cannot open the cursor a second time.

Globally Affecting Cursors with SET AUTOFREE

If you do not specify a *cursor_id* or *cursor_id_var*, the SET AUTOFREE statement affects all subsequent cursors in the program.

The following example shows how to enable the Autofree feature for all subsequent cursors:

```
EXEC SQL set autofree enabled;
```

Using the For Clause to Specify a Specific Cursor

If you specify a *cursor_id* or *cursor_id_var*, the SET AUTOFREE statement affects only the cursor that you specify after the FOR keyword.

This option allows you to override a global setting for all cursors. For example, if you issue a SET AUTOFREE ENABLED statement 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 cursor named **x1**:

```
EXEC SQL set autofree enabled;  
EXEC SQL set autofree disabled for x1;
```

Associated and Detached Statements

When a cursor is automatically freed, its associated prepared statement (or associated statement) is also freed.

The term *associated statement* has a special meaning in the context of the Autofree feature. A cursor is associated with a prepared statement if it is the first cursor that you declare with the prepared statement, or if it is the first cursor that you declare with the statement after the statement is detached.

The term *detached statement* has a special meaning in the context of the Autofree feature. A prepared statement is detached if you do not declare a cursor with the statement, or if the cursor with which the statement is associated has been freed.

If the Autofree feature is enabled for a cursor and the cursor has an associated prepared statement, the database server frees memory allocated to the prepared statement as well as the memory allocated for the cursor. Suppose that you enable the Autofree feature for the following cursor:

```
/*Cursor associated with a prepared statement */  
EXEC SQL prepare sel_stmt 'select * from customer';  
EXEC SQL declare sel_curs2 cursor for sel_stmt;
```

When the database server closes the **sel_curs2** cursor, it automatically performs the equivalent of the following FREE statements:

```
FREE sel_curs2;  
FREE sel_stmt;
```

Because the **sel_stmt** statement is freed automatically, you cannot declare a new cursor on it unless you prepare the statement again.

Closing Cursors Implicitly

A potential problem exists with cursors that have the Autofree feature enabled. In a non-ANSI-compliant database, if you do not close a cursor explicitly and then open it again, the cursor is closed implicitly. This implicit closing of the cursor triggers the Autofree feature. The second time the cursor is opened, the database server generates an error message (`cursor not found`) because the cursor is already freed.

References

Related statements: CLOSE, DECLARE, FETCH, FREE, OPEN, and PREPARE

For more information on the Autofree feature, see the [*INFORMIX-ESQL/C Programmer's Manual*](#).

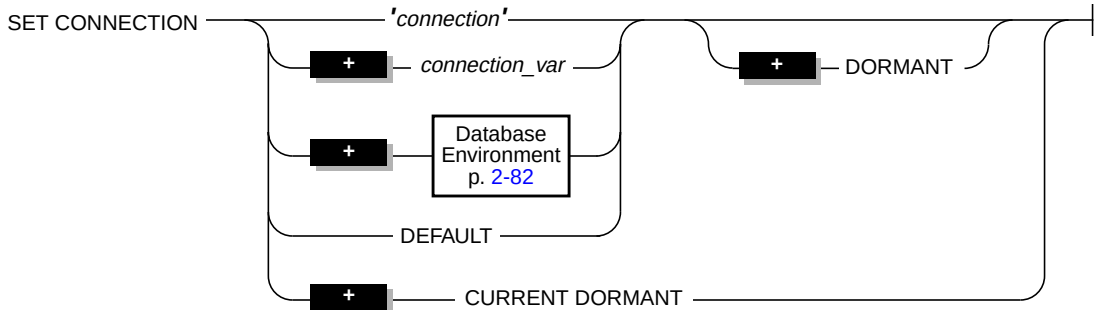
E/C

SET CONNECTION

Use the SET CONNECTION statement to reestablish a connection between an application and a database environment and make the connection current. You can also use the SET CONNECTION statement with the DORMANT option to put the current connection in a dormant state.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>connection</i>	Quoted string that identifies the name that you assigned to a specific connection It is the <i>connection</i> 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, <i>connection</i> must represent the current connection. If you use the SET CONNECTION statement without the DORMANT option, <i>connection</i> must represent a dormant connection.	Quoted String, p. 4-157
<i>connection_var</i>	Host variable that contains the value of <i>connection</i>	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
- Make the current connection dormant

You cannot use the SET CONNECTION statement in the statement text of a prepared statement.

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 is 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 2-79](#).) 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 Environment

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, a single-threaded environment can have only one active connection while the program executes. A thread-safe environment can have many threads (concurrent pieces of work performing particular tasks) in one ESQL/C application, and each thread can have one active connection.

An active connection is associated with a particular thread. Two threads cannot share the same active connection. Once a thread makes an active connection dormant, that connection is available to other threads. A dormant connection is still established but is not currently associated with any thread. For example, if the connection named `con1` is active in the thread named `thread_1`, the thread named `thread_2` cannot make connection `con1` its active connection until `thread_1` has made connection `con1` dormant.

The following code fragment from a thread-safe ESQL/C program shows how a particular thread within a thread-safe application makes a connection active, performs work on a table through this connection, and then makes the connection dormant so that other threads can use the connection:

```
thread_2()
{
    /* Make con2 an active connection */
    EXEC SQL connect to 'db2' as 'con2';
    /*Do insert on table t2 in db2*/
    EXEC SQL insert into table t2 values(10);
    /* make con2 available to other threads */
    EXEC SQL set connection 'con2' dormant;
}
.
.
.
```

If a connection to a database environment is initiated with the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, any thread that subsequently connects to that database environment can use an ongoing transaction. In addition, if an open cursor is associated with such a connection, the cursor remains open when the connection is made dormant. Threads within a thread-safe ESQL/C application can use the same cursor by making the associated connection current even though only one thread can use the connection at any given time.

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 a connection name in the initial CONNECT statement, you must use a database environment (such as a database name or a database pathname) as the connection name. For example, the following SET CONNECTION statement uses a database environment for the connection name because the CONNECT statement does not use a connection name. For information about quoted strings that contain a database environment, see [“Database Environment” on page 2-82](#).

```
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 established with the **DATABASE** or **CREATE 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. For more information, see [“DEFAULT Option” on page 2-79](#) and [“The Implicit Connection with DATABASE Statements” on page 2-80](#).

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 Dynamic Server Prior to Version 6.0

If the current connection is to a version of Dynamic Server earlier than 6.0, the following conditions apply when a SET CONNECTION statement executes:

- If the connection to be made dormant was established with 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

Related statements: CONNECT, DISCONNECT, and DATABASE

For a discussion of the SET CONNECTION statement and thread-safe applications, see the [*INFORMIX-ESQL/C Programmer's Manual*](#).

+

IDS

SET Database Object Mode

Use the SET Database Object Mode statement to change the mode of constraints, indexes, and triggers.

You can use this statement only with Dynamic Server.

Syntax

SET

Table-Mode
Format
p. 2-513

List-Mode
Format
p. 2-514

Usage

When you change the mode of constraints, indexes, or triggers, the change is persistent. The setting remains in effect until you change the mode of the database object again.

The **sysobjstate** system catalog table lists all of the database objects in the database and the current mode of each database object. For information on the **sysobjstate** system catalog table, see the [Informix Guide to SQL: Reference](#).

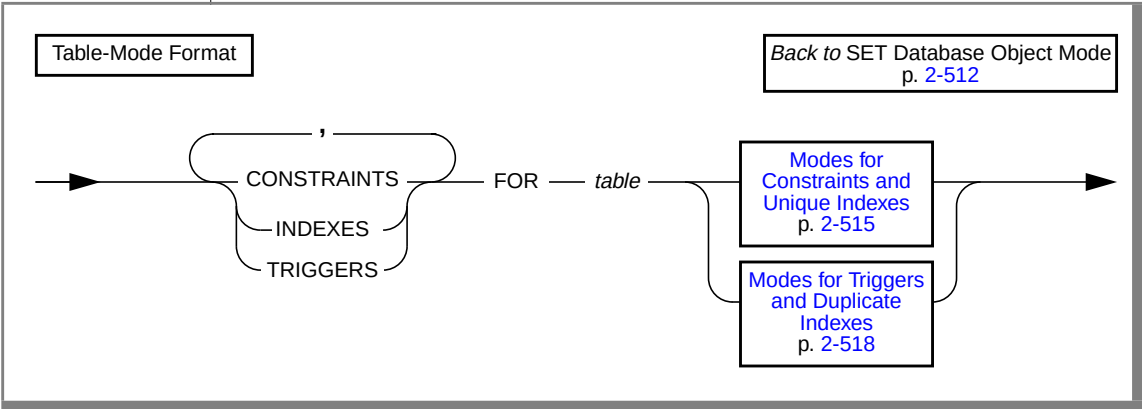
Privileges Required for Changing Database Object Modes

To change the mode of a constraint, index, or trigger, you must have the necessary privileges. 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 database object is defined and must have the Resource privilege on the database.
- You must have the Alter privilege on the table on which the database object is defined and the Resource privilege on the database.

Table-Mode Format

Use the table-mode format to change the mode of all database objects of a given type that have been defined on a particular table.



Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of the table on which the database objects reside	The table must be a local table. You cannot set database objects defined on a temporary table to disabled or filtering modes.	Database Object Name, p. 4-25

For example, to disable all constraints that are defined on the **cust_subset** table, enter the following statement:

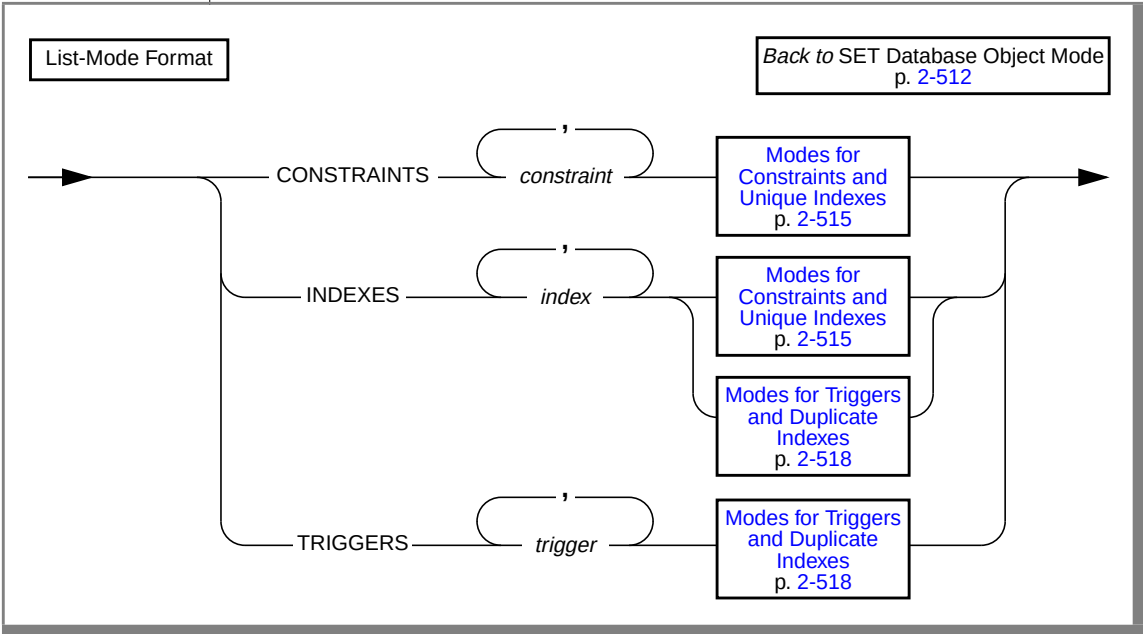
```
SET CONSTRAINTS FOR cust_subset DISABLED
```

When you use the table-mode format, you can change the modes of more than one database-object type with a single SET Database Object Mode statement. For example, to enable all constraints, indexes, and triggers that are defined on the **cust_subset** table, enter the following statement:

```
SET CONSTRAINTS, INDEXES, TRIGGERS FOR cust_subset  
ENABLED
```


List-Mode Format

Use the list-mode format to change the mode for a particular constraint, index, or trigger.



Element	Purpose	Restrictions	Syntax
<i>constraint</i>	Name of the constraint whose mode is to be set	Each constraint in the list must be a local constraint. All constraints in the list must be defined on the same table.	Database Object Name, p. 4-25
<i>index</i>	Name of the index whose mode is to be set	Each index in the list must be a local index. All indexes in the list must be defined on the same table.	Database Object Name, p. 4-25
<i>trigger</i>	Name of the trigger whose mode is to be set	Each trigger in the list must be a local trigger. All triggers in the list must be defined on the same table.	Database Object Name, p. 4-25

For example, to change the mode of the unique index **unq_ssn** on the **cust_subset** table to filtering, enter the following statement:

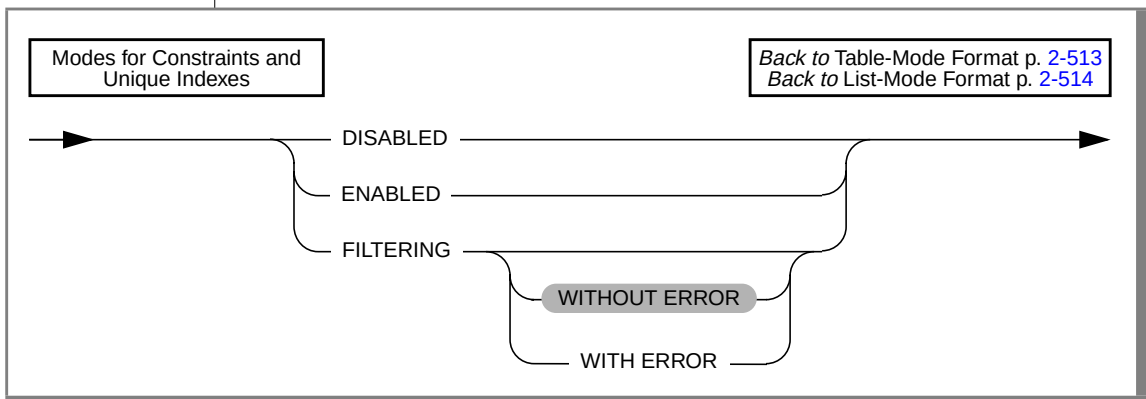
```
SET INDEXES unq_ssn FILTERING
```

You can also use the list-mode format to change the mode for a list of constraints, indexes, or triggers that are defined on the same table. Assume that four triggers are defined on the **cust_subset** table: **insert_trig**, **update_trig**, **delete_trig**, and **execute_trig**. Also assume that all four triggers are enabled. To disable all triggers except **execute_trig**, enter the following statement:

```
SET TRIGGERS insert_trig, update_trig, delete_trig DISABLED
```

Modes for Constraints and Unique Indexes

You can specify a disabled, enabled, or filtering mode for a constraint or a unique index.



If you do not specify the mode for a constraint in a CREATE TABLE, ALTER TABLE, or SET Database Object Mode statement, the constraint is enabled by default.

If you do not specify the mode for a unique index in the CREATE INDEX or SET Database Object Mode statement, the unique index is enabled by default.

For definitions of the disabled, enabled, and filtering modes, see [“Using Database Object Modes with Data Manipulation Statements” on page 2-518](#). For an explanation of the benefits of these modes, see [“Benefits of Database Object Modes” on page 2-531](#).

Error Options for Filtering Mode

When you change the mode of a constraint or unique index to filtering, you can specify the following error options: `WITHOUT ERROR` or `WITH ERROR`. These error options control whether the database server displays an integrity-violation error message after it executes these statements.

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, the database server returns an integrity-violation error message.

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 do not apply when you attempt to change the mode of a disabled constraint or disabled unique index to the enabled or filtering mode, and the SET Database Object Mode 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 is 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 Database Object Mode statement.

Violations and Diagnostics Tables for Filtering Mode

When you set database objects to filtering, be sure to start the violations and diagnostics tables for the target table on which the filtering-mode, database 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

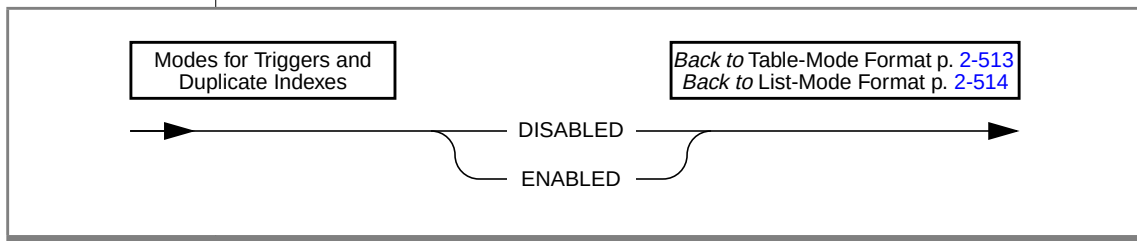
If you have not started a violations and a diagnostics table for the table that has database objects set to filtering, the database server returns an error when the first INSERT, DELETE, or UPDATE statement fails to satisfy an integrity requirement on the table.

To prevent this error, use the START VIOLATIONS TABLE statement to start the violations and diagnostics tables for the target table at one of the following points:

- Before you set any database objects that are defined on the table to the filtering mode
- After you set database objects to filtering, but before any users issue INSERT, DELETE, or UPDATE statements that might violate integrity requirements on the target table

Modes for Triggers and Duplicate Indexes

You can specify the disabled or enabled modes for triggers or duplicate indexes



If you do not specify the mode for a trigger in the CREATE TRIGGER or SET Database Object Mode statement, the trigger is enabled by default.

If you do not specify the mode for a duplicate index in the CREATE INDEX or SET Database Object Mode statement, the duplicate index is enabled by default.

For definitions of the disabled and enabled modes, see [“Using Database Object Modes with Data Manipulation Statements”](#). For an explanation of the benefits of these two modes, see [“Benefits of Database Object Modes” on page 2-531](#).

Using Database Object Modes with Data Manipulation Statements

You can use database object modes to control the effects of INSERT, DELETE, and UPDATE statements. Your choice of mode affects the tables whose data you are manipulating, the behavior of the database objects defined on those tables, and the behavior of the data manipulation statements themselves.

Definition of Enabled Mode

Constraints, indexes, and triggers are enabled by default. When a database object is enabled, the database server recognizes the existence of the database object and takes the database object into consideration while it executes 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 a database object is disabled, the database server acts as if the database 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 a database object is in the filtering mode, the database object behaves the same as in the enabled mode in that the database server recognizes the existence of the database 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 database 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 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 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 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_reowner
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 2-517](#). For further information on the structure of the violations and diagnostics tables, see the START VIOLATIONS TABLE statement on [page 2-594](#).

Using Modes to Achieve Data Integrity

In addition to using modes with data manipulation statements, you can also use modes when you add a new constraint or new unique index to a target table. When you select the correct 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 Database Object Mode statement to switch the 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 Database Object Mode 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 Modes to Achieve Data Integrity

The following example shows how to use database 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 Database 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 Database 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 database 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 Database Object Mode Statement to Capture Violations

Now that violations and diagnostics tables exist for the target table, user **joe** issues the following SET Database Object Mode 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 Database Object Mode 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 Database Object Mode 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 Database Object Mode statement.

ssn	fname	lname	city	informix_tupleid	informix_optype	informix_reowner
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 Database Object Mode statement.
- The value joe in the **informix_reowner** 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 the contents of the **cust_subset_dia** diagnostics table after user **joe** issues the SET Database Object Mode 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 database 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 Database Objects

Once all the nonconforming rows in the **cust_subset** table are corrected, user **joe** issues the following SET Database Object Mode statement to set the new constraint and index to the enabled mode:

```
SET CONSTRAINTS, INDEXES FOR cust_subset ENABLED
```

This time the SET Database Object Mode 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 Database Object Modes

The preceding examples show how database 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 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 database 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 Database Object Mode statement that switches the disabled database 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 Database Object Mode statement to switch the disabled database objects to the enabled or filtering mode.

Benefits of Enabled Mode

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.

References

Related statements: START VIOLATIONS TABLE and STOP VIOLATIONS TABLE

For a discussion of modes violation detection, see the [Informix Guide to SQL: Tutorial](#).

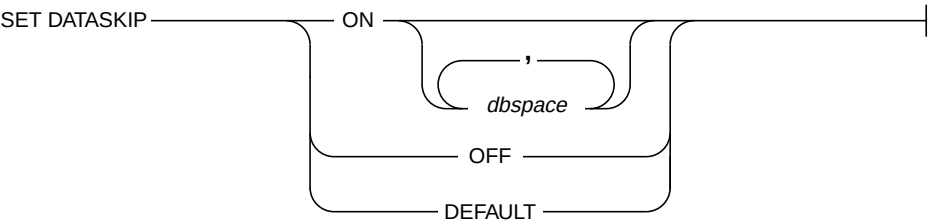
For information on the system catalog tables associated with the SET Database Object Mode statement, see the **sysobjstate** and **sysviolations** tables in the [Informix Guide to SQL: Reference](#).

+

SET DATASKIP

The SET DATASKIP statement allows you to control whether the database 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
<i>dbspace</i>	Name of the skipped dbspace	The dbspace must exist at the time the statement is executed.	Identifier, p. 4-113

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.

E/C

In ESQL/C, 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 setting of the dataskip feature can be changed at runtime.

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 into any available *dbspace*. When no *dbspace* is available, an error is returned.

- Indexing

When you perform updates that affect the index, such as when you insert or delete records, or when you update an indexed field, the index must be available.

When you try to create an index, the dbspace you want to use must be available.

- Serial keys

The first fragment is used to store the current serial-key value internally. This is not visible to you except when the first fragment becomes unavailable and a new serial key value is required, which happens during insert statements.

References

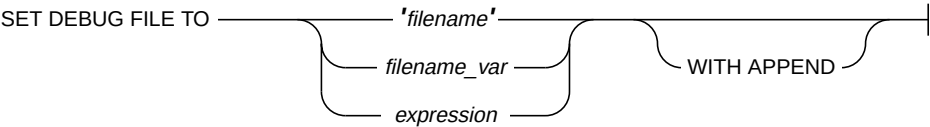
For additional information about the dataskip feature, see your [*Administrator's Guide*](#).

+

SET DEBUG FILE TO

Use the SET DEBUG FILE TO statement to name the file that is to hold the run-time trace output of a stored procedure.

Syntax



Element	Purpose	Restrictions	Syntax
<i>expression</i>	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 <i>filename</i> parameter.	Expression, p. 4-33
<i>filename</i>	Quoted string that identifies the pathname of the file that contains the output of the TRACE statement For information on the default actions that are taken if you omit the pathname, see “Location of the Output File” on page 2-539.	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. For further information, see “Using the WITH APPEND Option” on page 2-538.	Quoted String, p. 4-157 . Name must conform to the naming conventions of your operating system.
<i>filename_var</i>	Host variable that holds the value of <i>filename</i>	The host variable must be a character data type.	Name 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 **debug.out**:

```
SET DEBUG FILE TO 'debug' || '.out'
```

References

Related statement: TRACE

For a task-oriented discussion of stored procedures, see the [Informix Guide to SQL: Tutorial](#).

+

IDS

E/C

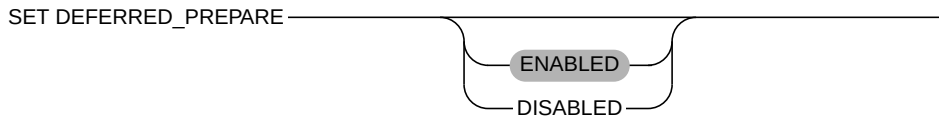
SET DEFERRED_PREPARE

Use the SET DEFERRED_PREPARE statement to defer sending a PREPARE statement to the database server until the OPEN or EXECUTE statement is sent.

You can use this statement only with Dynamic Server.

Use this statement with ESQL/C.

Syntax



Usage

The SET DEFERRED_PREPARE statement causes the application program to delay sending the PREPARE statement to the database server until the OPEN or EXECUTE statement is executed. In effect, the PREPARE statement is bundled with the other statement so that one round trip of messages instead of two is sent between the client and the server.

The Deferred-Prepare feature works with the following sequences:

- PREPARE, DECLARE, OPEN statement blocks that operate with the FETCH or PUT statements
- PREPARE, EXECUTE statement blocks or the EXECUTE IMMEDIATE statement

You can also set the **IFX_DEFERRED_PREPARE** environment variable to enable the Deferred-Prepare feature for all prepared statements in a program.

SET DEFERRED_PREPARE Options

You can specify the `ENABLED` or `DISABLED` options for the `SET DEFERRED_PREPARE` statement. If you do not specify either option, the default is `ENABLED`. The following example shows how to enable the Deferred-Prepare feature by default:

```
EXEC SQL set deferred_prepare;
```

ENABLED Option

Use the `ENABLED` option to enable the Deferred-Prepare feature within the application. The following example shows how to use the `ENABLED` option:

```
EXEC SQL set deferred_prepare enabled;
```

When you enter a `SET DEFERRED_PREPARE ENABLED` statement in your application, the Deferred-Prepare feature is enabled for all `PREPARE` statements in the application. The application then exhibits the following behavior:

- The sequence `PREPARE`, `DECLARE`, `OPEN` sends the `PREPARE` statement to the database server with the `OPEN` statement.
- If a prepared statement contains syntax errors, the database server does not return error messages to the application until the application declares a cursor for the prepared statement and opens the cursor.
- The sequence `PREPARE`, `EXECUTE` sends the `PREPARE` statement to the database server with the `EXECUTE` statement. If a prepared statement contains syntax errors, the database server does not return error messages to the application until the application attempts to execute the prepared statement.

DESCRIBE Restriction with the ENABLED Option

If you use the Deferred-Prepare feature in a `PREPARE`, `DECLARE`, `OPEN` statement block that contains a `DESCRIBE` statement, the `DESCRIBE` statement must follow the `OPEN` statement rather than the `PREPARE` statement. If the `DESCRIBE` statement follows the `PREPARE` statement, the `DESCRIBE` statement results in an error.

DISABLED Option

Use the DISABLED option to disable the Deferred-Prepare feature within the application. The following example shows how to use the DISABLED option:

```
EXEC SQL set deferred_prepare disabled;
```

When you disable the Deferred-Prepare feature, the application sends each PREPARE statement to the database server when the PREPARE statement is executed.

Example of SET DEFERRED_PREPARE

The following code fragment shows a SET DEFERRED PREPARE statement with a PREPARE, EXECUTE statement block. In this case, the database server executes the PREPARE and EXECUTE statements all at once.

```
EXEC SQL BEGIN DECLARE SECTION;
      int a;
EXEC SQL END DECLARE SECTION;
EXEC SQL allocate descriptor 'desc';
EXEC SQL create database test;
EXEC SQL create table x (a int);

/* Enable Deferred-Prepare feature */
EXEC SQL set deferred_prepare enabled;

/* Prepare an INSERT statement */
EXEC SQL prepare ins_stmt from 'insert into x values(?)';

a = 2;
EXEC SQL EXECUTE ins_stmt using :a;
if (SQLCODE)
    printf("EXECUTE : SQLCODE is %d\n", SQLCODE);
```

Using Deferred-Prepare with OPTOFC

You can use the Deferred-Prepare and OPTOFC (Open-Fetch-Close Optimization) features together in your application. The OPTOFC feature delays sending the OPEN message to the database server until the FETCH message is sent.

The following situations occur if you enable the Deferred-Prepare and OPTOFC features at the same time:

- If the text of a prepared statement contains syntax errors, the error messages are not returned to the application until the first FETCH statement is executed.
- A DESCRIBE statement cannot be executed until after the FETCH statement.
- You must issue an ALLOCATE DESCRIPTOR statement before a DESCRIBE or GET DESCRIPTOR statement can be executed.

The database server performs an internal execution of a SET DESCRIPTOR statement which sets the TYPE, LENGTH, DATA, and other fields in the system descriptor area. You can specify a GET DESCRIPTOR statement after the FETCH statement to see the data that is returned.

References

Related statements: DECLARE, DESCRIBE, EXECUTE, OPEN, and PREPARE

For a task-oriented discussion of the PREPARE statement and dynamic SQL, see the [Informix Guide to SQL: Tutorial](#).

For more information about concepts relating to the SET DEFERRED_PREPARE statement, see the [INFORMIX-ESQL/C Programmer's Manual](#).

For more information on the **IFX_DEFERRED_PREPARE** environment variable, see the [Informix Guide to SQL: Reference](#).

+

E/C

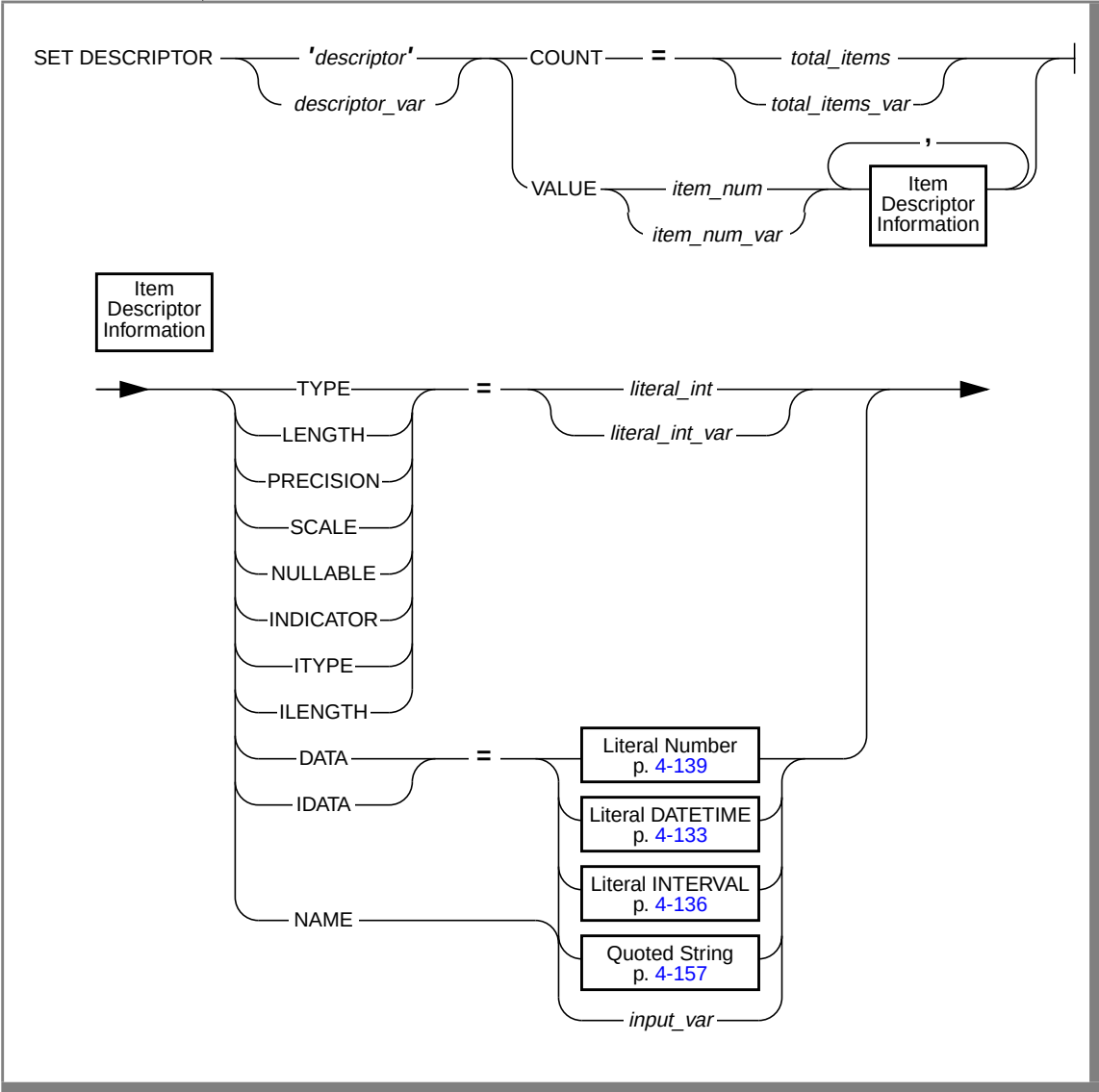
SET DESCRIPTOR

Use the SET DESCRIPTOR statement to 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 (typically the items are in a WHERE clause)
- To set the item descriptor fields for each value for which you are providing descriptions in the system-descriptor area (typically the items are in a WHERE clause)
- To modify the contents of an item-descriptor field after you use the DESCRIBE statement to fill the fields for a SELECT or an INSERT statement

Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>descriptor</i>	String that identifies the 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. 4-157
<i>descriptor_var</i>	Host variable that holds the value of <i>descriptor</i>	The same restrictions apply to <i>descriptor_var</i> as apply to <i>descriptor</i> .	Name must conform to language-specific rules for variable names.
<i>input_var</i>	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 <i>input_var</i> must be appropriate for the specified field.	Name must conform to language-specific rules for variable names.
<i>item_num</i>	Unsigned integer that specifies one of the item descriptors in the system-descriptor area	The value of <i>item_num</i> must be greater than 0 and less than (or equal to) the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.	Literal Number, p. 4-139
<i>item_num_var</i>	Host variable that holds the value of <i>item_num</i>	The same restrictions apply to <i>item_num_var</i> as apply to <i>item_num</i> .	Name must conform to language-specific rules for variable names.
<i>literal_int</i>	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 <i>literal_int</i> 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 2-548 . For the restrictions that apply to other field types, see the individual headings for field types under “Using the VALUE Clause” on page 2-548 .	Literal Number, p. 4-139

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>literal_int_var</i>	Host variable that contains the value of <i>literal_int</i>	The same restrictions apply to <i>literal_int_var</i> as apply to <i>literal_int</i> .	Name must conform to language-specific rules for variable names.
<i>total_items</i>	Literal integer that specifies how many items are actually described in the system-descriptor area	The integer that <i>value</i> specifies must be greater than 0 and less than (or equal to) the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.	Literal Number, p. 4-139
<i>total_items_var</i>	Host variable that holds a literal integer that specifies how many items are actually described in the system-descriptor area	The same restrictions apply to <i>total_items_var</i> as apply to <i>total_items</i> .	Name must conform to language-specific rules for variable names.

(2 of 2)

Usage

Use the SET DESCRIPTOR statement to assign values to a system-descriptor area.

If an error occurs during the assignment to any identified system-descriptor fields, the contents of all identified fields are set to 0 or null, depending on the variable type.

Using the COUNT Clause

Use the COUNT clause to set the number of items that are to be used in the system-descriptor area.

If you allocate a system-descriptor area with more items than you are using, you need to set the COUNT field to the number of items that you are actually using. The following example shows the sequence of statements in 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;
```

Using the 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 that the database server described during a DESCRIBE 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

For code that is easier to maintain, use the predefined constants for these SQL data types instead of their actual integer values. These constants are defined in the **sqltypes.h** header file.

The following example shows how you can set the TYPE field in ESQL/C:

```
main()
{
EXEC SQL BEGIN DECLARE SECTION;
    int itemno, type;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL allocate descriptor 'desc1' with max 5;
...
EXEC SQL set descriptor 'desc1' value 2 type = 5;

type = 2; itemno = 3;
EXEC SQL set descriptor 'desc1' value :itemno type = :type;
}
```

Compiling Without the -xopen Option

If you compile without 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.

X/O

Setting the TYPE Field in X/Open Programs

In X/Open mode, you must use the X/Open set of integer codes for the data type in the TYPE field. 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.

Setting the DATA or IDATA Field

When you set the **DATA** or **IDATA** 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** or **IDATA** field for an item without setting TYPE for that item. If you set the **TYPE** field for an item to a character type, you must also set the **LENGTH** field. If you do not set the **LENGTH** field for a character item, you receive an error.

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, **LENGTH** is automatically set to the maximum length of the character field that is specified in your table.

This information is identical for **ILENGTH**.

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 Field

If you want to put a null value into the system-descriptor area, set the **INDICATOR** field to -1, and do not set the **DATA** field.

If you set the **INDICATOR** field to 0 to indicate that the data is not null, you must set the **DATA** field.

Setting the ITYPE 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 2-548](#).

References

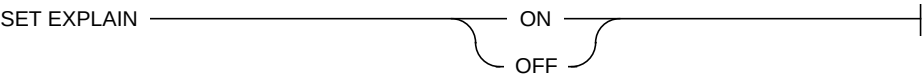
Related statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, and PUT

For more information on system-descriptor areas, refer to the [INFORMIX-ESQL/C Programmer's Manual](#).

SET EXPLAIN

Use the SET EXPLAIN statement to display the query plan the optimizer chooses, an estimate of the number of rows returned, and a relative cost of the query.

Syntax



Usage

The SET EXPLAIN statement provides various measurements of the work involved in performing a query. The ON option causes measurements for each subsequent query to be generated and written to an output file. The OFF option terminates the SET EXPLAIN statement so that measurements for subsequent queries are no longer generated or written to the output file.

The SET EXPLAIN ON statement remains in effect until you issue a SET EXPLAIN OFF statement or until the program ends.

If the output file does not exist when you issue the SET EXPLAIN ON statement, the database server creates the output file. If the output file already exists when you issue the SET EXPLAIN ON statement, subsequent output is appended to the file.

If you do not enter a SET EXPLAIN statement, the default behavior is OFF. The database server does not generate measurements for queries.

Execution of the SET EXPLAIN Statement

The SET EXPLAIN ON statement executes during the database server optimization phase, which occurs when you initiate a query. For queries that are associated with a cursor, if the query is prepared and does not have host variables, optimization occurs when you prepare it. Otherwise, optimization occurs when you open the cursor.

UNIX

Name and Location of the Output File

On UNIX, when you issue a SET EXPLAIN ON statement, the plan that the optimizer chooses for each subsequent query is written to the **sqexplain.out** file. The owner name (for example, *owner.customer*) qualifies table names in the **sqexplain.out** file.

If the client application and the database server are on the same computer, the **sqexplain.out** file is stored in your current directory. If you are using a version 5.x or earlier client application and the **sqexplain.out** file does not appear in the current directory, check your home directory for the file.

When the current database is on another computer, the **sqexplain.out** file is stored in your home directory on the remote host. ♦

WIN NT

On Windows NT, when you issue a SET EXPLAIN ON statement, the plan that the optimizer chooses for each subsequent query is written to the file **%INFORMIXDIR%\sqexpln\username.out**. The owner name (for example, *owner.customer*) qualifies table names in the file. ♦

SET EXPLAIN Output

The SET EXPLAIN output file contains a copy of the query, a query plan 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.

Effects of Directing Optimization on SET EXPLAIN Output

When you use optimizer directives or the SET OPTIMIZATION statement to alter a query plan, the alterations are recorded in the output of the SET EXPLAIN statement. For example, if you SET OPTIMIZATION to LOW, the output of SET EXPLAIN displays the following uppercase string as the first line:

```
QUERY:{LOW}
```

If you SET OPTIMIZATION to HIGH, the output of SET EXPLAIN displays the following uppercase string as the first line:

```
QUERY:
```

In Dynamic Server, if you provide optimizer directives for your query, the output of SET EXPLAIN displays the following lines immediately after the query:

```
DIRECTIVES FOLLOWED:  
[List of dierctives specified]
```



References

Related statements: SET OPTIMIZATION and UPDATE STATISTICS

For discussions of SET EXPLAIN and of analyzing the output of the optimizer, see your [Performance Guide](#).

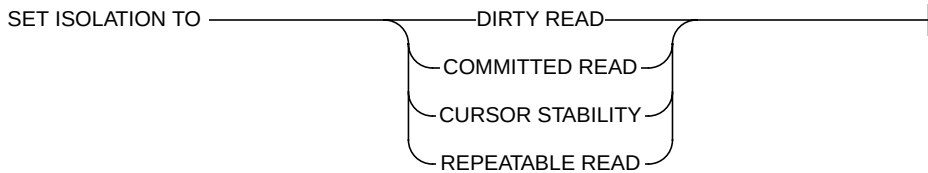
+

SET ISOLATION

Use the SET ISOLATION statement 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. For a comparison of these two statements, see the SET TRANSACTION statement on [page 2-584](#).

Syntax



Usage

The database isolation level affects read concurrency when rows are retrieved from the database. The database server uses shared locks to support different levels of isolation among processes attempting to access data.

The update or delete process always acquires an exclusive lock on the row that is being modified. The level of isolation does not interfere with rows that you are updating or deleting. If another process attempts to update or delete rows that you are reading with an isolation level of Repeatable Read, that process is denied access to those rows.

In ESQL/C, 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 <i>phantom</i> 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, the database 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 or insert a row that meets the search criteria of your query during your transaction. If you repeat the query during the transaction, you reread the same information. The shared locks are released only when the transaction commits or rolls back. Repeatable Read is the default isolation level in an ANSI-compliant database.

(2 of 2)

In the following example, the user sets the isolation level to Repeatable Read so that a query rereads the same information if it is repeated during the transaction:

```
SET ISOLATION TO REPEATABLE READ
```

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 retrieval of a BYTE or TEXT column can vary, depending on the database isolation level. Under Dirty Read or Committed Read levels of isolation, a process is permitted to read a BYTE or TEXT column that is either deleted (if the delete is not yet committed) or in the process of being deleted. Under these isolation levels, an application can read a deleted data when certain conditions exist. For information about these conditions, see your [*Administrator's Guide*](#).

DB

When you use DB-Access, as you use higher levels of isolation, lock conflicts occur more frequently. For example, if you use Cursor Stability, more lock conflicts occur than if you use Committed Read. ♦

E/C

In ESQL/C, if you use a scroll cursor in a transaction, you can force consistency between your temporary table and the database table either by setting the isolation level to 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

Related statements: CREATE DATABASE, SET LOCK MODE, and SET TRANSACTION

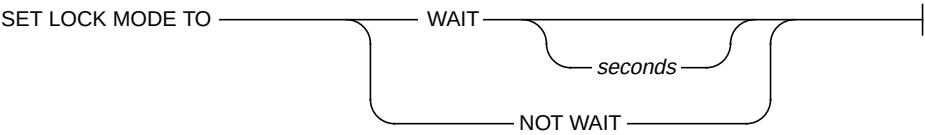
For a discussion of setting the isolation level, see the [Informix Guide to SQL: Tutorial](#).

+

SET LOCK MODE

Use the SET LOCK MODE statement to define how the database server handles a process that tries to access a locked row or table.

Syntax



Element	Purpose	Restrictions	Syntax
<i>seconds</i>	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. If you specify a value for <i>seconds</i>, the value applies only when the waiting period is shorter than the system default.	Literal Number, p. 4-139

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	The database server ends the operation immediately and returns an error code. This condition is the default.
WAIT	The database server suspends the process until the lock releases.
WAIT <i>seconds</i>	The database server 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, the database server ends the operation and returns an error code.

In the following example, the user specifies that the process should be suspended until the lock is released:

```
SET LOCK MODE TO WAIT
```

In the following example, the user specifies that if the process requests a locked row the operation should end immediately and an error code should be returned:

```
SET LOCK MODE TO NOT WAIT
```

In the following example, the user places an upper limit of 17 seconds on the length of any wait:

```
SET LOCK MODE TO WAIT 17
```

WAIT Clause

The WAIT clause causes the database server to suspend the process until the lock is released or until a specified number of seconds have passed without the lock being released.

The database server protects against the possibility of a deadlock when you request the WAIT option. Before the database server suspends a process, it checks whether suspending the process could create a deadlock. If the database server discovers that a deadlock could occur, it ends the operation (overruling your instruction to wait) and returns an error code. In the case of either a suspected or actual deadlock, the database server returns an error.

Cautiously use the unlimited waiting period that was created when you specify the WAIT option without *seconds*. If you do not specify an upper limit, and the process that placed the lock somehow fails to release it, suspended processes could wait indefinitely. Because a true deadlock situation does not exist, the database server does not take corrective action.

In a networked environment, the DBA uses the ONCONFIG parameter DEADLOCK_TIMEOUT to establish a default value for *seconds*. If you use a SET LOCK MODE statement to set an upper limit, your value applies only when your waiting period is shorter than the system default. The number of seconds that the process waits applies only if you acquire locks within the current database server and a remote database server within the same transaction.

References

Related statements: LOCK TABLE, SET ISOLATION, SET TRANSACTION and UNLOCK TABLE

For a discussion on how to set the lock mode, see the [Informix Guide to SQL: Tutorial](#).

Use the SET LOG statement to change your database logging mode from buffered transaction logging to unbuffered transaction logging or vice versa.

Diagram illustrating a buffered SET-LOG interface. A horizontal line represents the interface, with 'SET' on the left and 'LOG' on the right. A bracket below the line, spanning a central portion, is labeled 'BUFFERED'.

You activate transaction logging when you create a database or add logging to an existing database. These transaction logs can be buffered or unbuffered.

Buffered logging is a type of logging that holds transactions in a memory buffer until the buffer is full, regardless of when the transaction is committed or rolled back. The database server provides this option to speed up operations by reducing the number of disk writes. 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 in the following example changes the transaction logging mode to buffered logging:

SET BUFFERED LOG

Unbuffered logging is a type of logging that does not hold transactions in a memory buffer. As soon as a transaction ends, the database server writes the transaction to disk. If a system failure occurs when you are using unbuffered logging, you recover all completed transactions. The default condition for transaction logs is unbuffered logging.

The SET LOG statement in the following example changes the transaction logging mode to unbuffered logging:

```
SET LOG
```

The SET LOG statement redefines the mode for the current session only. The default mode, which the database administrator sets with the **ondblog** utility, remains unchanged.

The buffering option does not affect retrievals from external tables. For distributed queries, a database with logging can retrieve only from databases with logging, but it makes no difference whether the databases use buffered or unbuffered logging.

ANSI

An ANSI-compliant database cannot use buffered logs.

You cannot change the logging mode of ANSI-compliant databases. If you created a database with the WITH LOG MODE ANSI keywords, you cannot later use the SET LOG statement to change the logging mode to buffered or unbuffered transaction logging. ♦

References

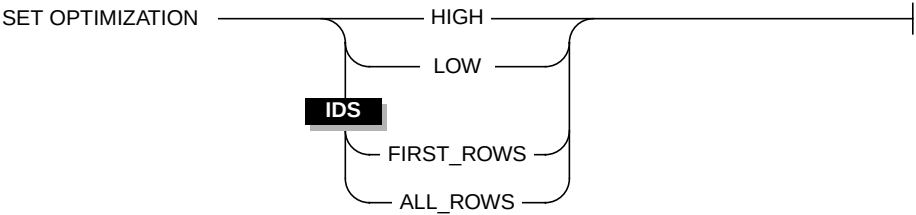
Related statement: CREATE DATABASE

+

SET OPTIMIZATION

Use the SET OPTIMIZATION statement to specify the time the optimizer spends to determine the query plan or to specify the optimization goals of the query.

Syntax



Usage

You can execute a SET OPTIMIZATION statement at any time. The optimization level carries across databases on the current database server.

When you issue a SET OPTIMIZATION statement, the option that you specify is persistent. That is, the new optimization level remains in effect until you issue another SET OPTIMIZATION statement or until the program ends.

The default database-server optimization level for the time the optimizer spends determining the query plan is HIGH.

In Dynamic Server, the default database-server optimization level for the optimization goal of the query is ALL_ROWS.

Although you can set only one option at a time, you can issue two SET OPTIMIZATION statements: one that specifies the time the optimizer spends to determine the query plan and one that specifies the optimization goal of the query. ♦

IDS

HIGH and LOW Options

The HIGH and LOW options relate to the time the optimizer spends to determine the query plan:

- **HIGH**

This option directs the optimizer to use a sophisticated, cost-based algorithm that examines all reasonable query-plan choices and selects the best overall alternative.

For large joins, this algorithm can incur more overhead than you desire. In extreme cases, you can run out of memory.

- **LOW**

This option directs the optimizer to use a less sophisticated, but faster, optimization algorithm. This algorithm eliminates unlikely join strategies during the early stages of optimization and reduces the time and resources spent during optimization.

When you specify a low level of optimization, the database server might not select the optimal strategy because the strategy was eliminated from consideration during the early stages of the algorithm.

IDS

FIRST_ROWS and ALL_ROWS Options

In Dynamic Server, FIRST_ROWS and ALL_ROWS options relate to the optimization goal of the query:

- **FIRST_ROWS**

This option directs the optimizer to choose the query plan that returns the first result record as soon as possible.

- ★ **ALL_ROWS**

This option directs the optimizer to choose the query plan which returns all the records as quickly as possible.

You can also specify the optimization goal of a specific query with the optimization-goal directive. For more information on how to use a directive to specify the optimization goal of a query, see [“Optimizer Directives” on page 4-141](#).

Optimizing Stored Procedures

For stored procedures that remain unchanged or change only slightly, you might want to set the SET OPTIMIZATION statement to HIGH when you create the procedure. This step stores the best query plans for the procedure. Then execute a SET OPTIMIZATION LOW statement before you execute the procedure. The procedure then uses the optimal query plans and runs at the more cost-effective rate.

Examples

The following example shows optimization across a network. The **central** database (on the **midstate** database server) is to have LOW optimization; the **western** database (on the **rockies** database server) is to have HIGH optimization.

```
CONNECT TO 'central@midstate';
SET OPTIMIZATION LOW;
SELECT * FROM customer;
CLOSE DATABASE;
CONNECT TO 'western@rockies';
SET OPTIMIZATION HIGH;
SELECT * FROM customer;
CLOSE DATABASE;
CONNECT TO 'wyoming@rockies';
SELECT * FROM customer;
```

The **wyoming** database is to have HIGH optimization because it resides on the same database server as the **western** database. The code does not need to respecify the optimization level for the **wyoming** database because the **wyoming** database resides on the **rockies** database server like the **western** database.

IDS

In Dynamic Server, the following example directs to use the most time to determine a query plan and to then return the first rows of the result as soon as possible.

```
SET OPTIMIZATION LOW;
SET OPTIMIZATION FIRST_ROWS;
SELECT lname, fname, bonus
FROM sales_emp, sales
WHERE sales.empid = sales_emp.empid AND bonus > 5,000
ORDER BY bonus DESC
```



References

Related statements: SET EXPLAIN and UPDATE STATISTICS

If you are using Dynamic Server, refer to “[Optimizer Directives](#)” on [page 4-141](#) for information on other methods by which you can alter the query plan of the optimizer. ♦

For more information on how to optimize queries, see your [Performance Guide](#).

+

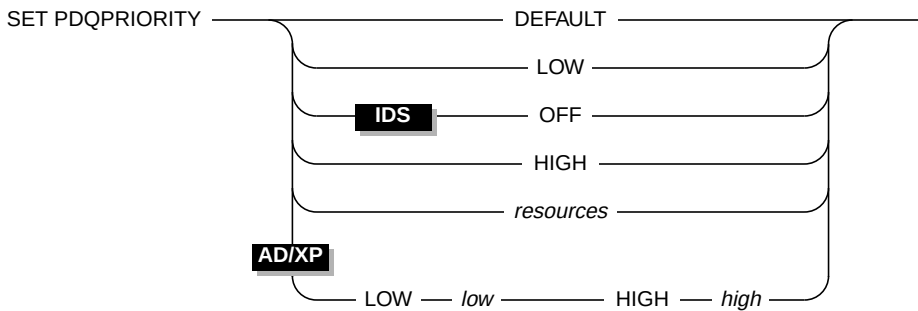
W/D

SET PDQPRIORITY

The SET PDQPRIORITY statement allows an application to set the query priority level dynamically within an application.

This statement is not available with Dynamic Server, Workgroup and Developer Editions. ♦

Syntax



Element	Purpose	Restrictions	Syntax
<i>high</i>	Integer value that specifies the desired resource allocation	You must specify a value in the range 1 to 100. The <i>high</i> value must be greater than or equal to the <i>low</i> value.	Literal Number, p. 4-139
<i>low</i>	Integer value that specifies the minimum acceptable resource allocation	You must specify a value in the range 1 to 100.	Literal Number, p. 4-139
<i>resources</i>	Integer value that specifies the query priority level and the percent of resources the database server uses to process the query	Value must be from -1 to 100. For information on the specific meanings of certain values, see “Allocating Database Server Resources” on page 2-571.	Literal Number, p. 4-139

Usage

The 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 database server uses for your queries.

For example, assume that the DBA sets the MAX_PDQPRIORITY parameter to 50. Then a user enters the following SET PDQPRIORITY statement to set the query priority level to 80 percent of resources.

```
SET PDQPRIORITY 80
```

When it processes the user's query, the database server uses the value of the MAX_PDQPRIORITY parameter to factor the query priority level set by the user. The database server silently processes the query with a priority level of 40. This priority level represents 50 percent of the 80 percent of resources that the user specifies.

IDS

In Dynamic Server, set PDQ priority to a value that is less than the quotient of 100 divided by the maximum number of prepared statements. For example, if two prepared statements are active, you should set PDQ priority to less than 50. ♦

AD/XP

In Dynamic Server with AD and XP Options, set PDQ priority to a value greater than 0 when you need more memory for database operations such as sorts, groups, and index builds. For guidelines on which values to use, see your [Performance Guide](#). ♦

SET PDQPRIORITY Keywords

The following table shows the keywords that you can enter for the SET PDQPRIORITY statement.

Keyword	Meaning
DEFAULT	Uses the value that is specified in the PDQPRIORITY environment variable
LOW	Signifies that data is fetched from fragmented tables in parallel In Dynamic Server, when you specify LOW, the database server uses no other forms of parallelism.
OFF	Indicates that PDQ is turned off (Dynamic Server only) The database server uses no parallelism. OFF is the default setting if you use neither the PDQPRIORITY environment variable nor the SET PDQPRIORITY statement.
HIGH	Signifies that the database server determines an appropriate value to use for PDQPRIORITY This decision is based on several factors, 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.

Allocating Database Server Resources

You can specify any integer in the range from -1 to 100 to indicate a query priority level as the percent of resources the database server uses to process the query.

Resources include the amount of memory and the number of processors. The higher the number you specify in this parameter, the more resources the database server uses. Although the use of more resources by a database server usually 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.

IDS

With the *resources* option, the following values are numeric equivalents of the keywords that indicate query priority level:

Value	Equivalent Keyword Priority Level
-1	DEFAULT
0	OFF (Dynamic Server only)
1	LOW (Dynamic Server only)

In Dynamic Server, the following statements are equivalent. The first statement uses the keyword **LOW** to establish a low query priority level. The second statement uses a value of 1 in the *resources* parameter to establish a low query-priority level.

```
SET PDQPRIORITY LOW;
```

```
SET PDQPRIORITY 1;
```

AD/XP

Using a Range of Values

In Dynamic Server with AD and XP Options, when you specify a range of values in **SET PDQPRIORITY**, you allow the Resource Grant Manager (RGM) some discretion when allocating resources. The largest value in the range is the desired resource allocation, while the smallest value is the minimum acceptable resource allocation for the query. If the minimum PDQ priority exceeds the available system resources, the RGM blocks the query. Otherwise, the RGM chooses the largest PDQ priority in the range specified in **SET PDQPRIORITY** that does not exceed available resources.

References

For information about configuration parameters and about the Resource Grant Manager, see your [Administrator's Guide](#) and your [Performance Guide](#).

For information about the **PDQPRIORITY** environment variable, see the [Informix Guide to SQL: Reference](#).

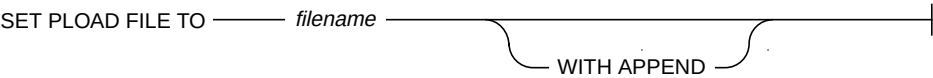
+

AD/XP

SET PLOAD FILE

Use the SET PLOAD FILE statement to prepare a log file for a session of loading or unloading data from or to an external table. The log file records summary statistics about each load or unload job. The log file also lists any reject files created during a load job.

You can use this statement only with Dynamic Server with AD and XP Options.



Element	Purpose	Restrictions	Syntax
<i>filename</i>	Name for the log file If you do not specify a log file name, log information is written to /dev/null .	If the file cannot be opened for writing, an error results.	Identifier p. 4-113

Usage

The WITH APPEND option allows you to append new log information to the existing log file.

Each time a session closes, the log file for that session also closes. If you issue more than one SET PLOAD FILE statement within a session, each new statement closes a previously opened log file and opens a new log file.

If you invoke a SET PLOAD FILE statement with a simple filename on a local database, the output file is located in your current directory. If your current database is on a remote database server, then the output file is located in your home directory on the remote database server, on the coserver where the initial connection was made. If you provide a full pathname for the file, it is placed in the directory and file specified on the remote server.

References

Related Statements: CREATE EXTERNAL TABLE

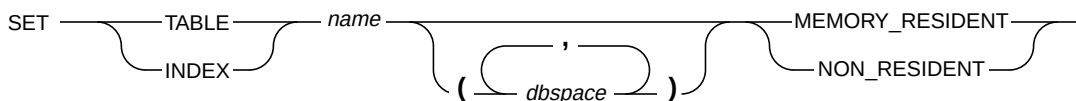
IDS

SET Residency

Use the SET Residency statement to specify that one or more fragments of a table or index be resident in shared memory as long as possible.

You can use this statement only with Dynamic Server.

Syntax



Element	Purpose	Restrictions	Syntax
<i>name</i>	Name of the table or index for which you want to change the resident state	The table or index must exist.	Database Object Name, p. 4-25
<i>dbspace</i>	Name of the dbspace in which the fragment resides	The dbspace must exist.	Identifier, p. 4-113

Usage

The SET Residency statement allows you to specify the tables, indexes, and data fragments that you want to remain in the buffer as long as possible. When a free buffer is requested, pages that are declared `MEMORY_RESIDENT` are considered last for page replacement.

The default resident state for database objects is nonresident. The resident state is persistent for the time the database server is up. That is, each time the database server is started you must specify the database objects that you want to remain in shared memory.

After a table, index, or data fragment is set to `MEMORY_RESIDENT`, the resident state remains in effect until one of the following events occurs:

- You use the SET Residency statement to set the database object to `NON_RESIDENT`.
- The database object is dropped.
- The database server is brought down.

You must be user **informix** to set or change the residency status of a database object.

Residency and the Changing Status of Fragments

If new fragments are added to a resident table, the fragments are not marked automatically as resident. You must issue the SET Residency statement for each new fragment or reissue the statement for the entire table.

Similarly, if a resident fragment is detached from a table, the residency status of the fragment remains unchanged. If you want the residency status to change to nonresident, you must issue the SET Residency statement to declare the specific fragment (or the entire table) as nonresident.

Examples

The following example shows how to set the residency status of an entire table.

```
SET TABLE tab1 MEMORY_RESIDENT
```

For fragmented tables or indexes, you can specify residency for individual fragments as the following example shows.

```
SET INDEX index1 (dbspace1, dbspace2) MEMORY_RESIDENT;  
SET TABLE tab1 (dbspace1) NON_RESIDENT
```

This example specifies that the **tab1** fragment in **dbspace1** is not to remain in shared memory while the **index1** fragments in **dbspace1** and **dbspace2** are to remain in shared memory as long as possible.

References

Related statements: ALTER FRAGMENT

For information on how to monitor the residency status of tables, indexes and fragments, refer to your [Administrator's Guide](#).

+

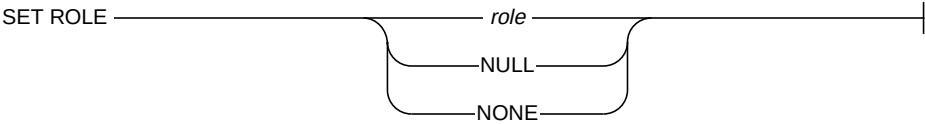
IDS

SET ROLE

Use the SET ROLE statement to enable the privileges of a role.

You can use this statement only with Dynamic Server.

Syntax



Element	Purpose	Restrictions	Syntax
<i>role</i>	Name of the role that you want to enable	The role must have been created with the CREATE ROLE statement.	Identifier, p. 4-113

Usage

Any user who is granted a role can enable the role using the SET ROLE statement. A user can only enable 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 database objects, such as tables, that were created during a session.

The scope of a role is within the current database only. A user cannot use the privileges acquired from a role to access data in another database. For example, if a user has privileges from a role in the database named **acctg**, and executes a distributed query over the databases named **acctg** and **inventory**, the user's query cannot access the data in the **inventory** database unless the user has been granted privileges in the **inventory** database.

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
        from projects
        where project_num > 100 and lname = 'Larkin';
printf ("%s is working on %s\n", efname, eproject);
EXEC SQL set role null;
```

References

Related statements: CREATE ROLE, DROP ROLE, GRANT, and REVOKE

For a discussion of how to use roles, see the [Informix Guide to SQL: Tutorial](#).

+

AD/XP

SET SCHEDULE LEVEL

The SET SCHEDULE LEVEL statement specifies the scheduling level of a query when queries are waiting to be processed.

You can use this statement only with Dynamic Server with AD and XP Options.

Syntax

SET SCHEDULE LEVEL *level*

Element	Purpose	Restrictions	Syntax
<i>level</i>	Integer value that determines the scheduling priority of a query	The value must be between 1 and 100. If the value falls outside the range of 1 and 100, the database server uses the default value of 50.	Literal Number, p. 4-139

Usage

The highest priority level is 100. That is, a query at level 100 is more important than a query at level 1. In general, the Resource Grant Manager (RGM) processes a query with a higher scheduling level before a query with a lower scheduling level. The exact behavior of the RGM is influenced by the setting of the DS_ADM_POLICY configuration parameter.

References

Related statement: SET PDQPRIORITY

For information about the Resource Grant Manager and DS_ADM_POLICY, refer to your [Administrator's Guide](#).

SET SESSION AUTHORIZATION

The SET SESSION AUTHORIZATION statement lets you change the user name under which database operations are performed in the current 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

SET SESSION AUTHORIZATION TO _____ 'user' _____|

Element	Purpose	Restrictions	Syntax
<i>user</i>	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. 4-113

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.

Restriction on Scope of SET SESSION AUTHORIZATION

When you assume the identity of another user by executing the SET SESSION AUTHORIZATION statement, you can perform operations in the current database only. You cannot perform an operation on a database object outside the current database, such as a remote table. In addition, you cannot execute a DROP DATABASE or RENAME DATABASE statement, even if the database is owned by the real or effective user.

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 customer TO mary;
SET SESSION AUTHORIZATION TO 'mary';
UPDATE customer
  SET fname = 'Carl'
  WHERE lname = 'Pauli';
```

ANSI

SET SESSION AUTHORIZATION and Transactions

If your database is not ANSI compliant, you must issue the SET SESSION AUTHORIZATION statement outside a transaction. If you issue the statement within a transaction, you receive an error message.

In an ANSI-compliant database, you can execute the SET SESSION AUTHORIZATION statement as long as you have not executed a statement that initiates an implicit transaction. Such statements either acquire locks or log data (for example, CREATE TABLE or SELECT). Statements that do not initiate an implicit transaction are statements that do not acquire locks or log data (for example, SET EXPLAIN and SET ISOLATION).

The COMMIT WORK and DATABASE statements do not initiate implicit transactions. So, in an ANSI-compliant database, you can execute the SET SESSION AUTHORIZATION statement immediately after a DATABASE statement or a COMMIT WORK statement. ♦

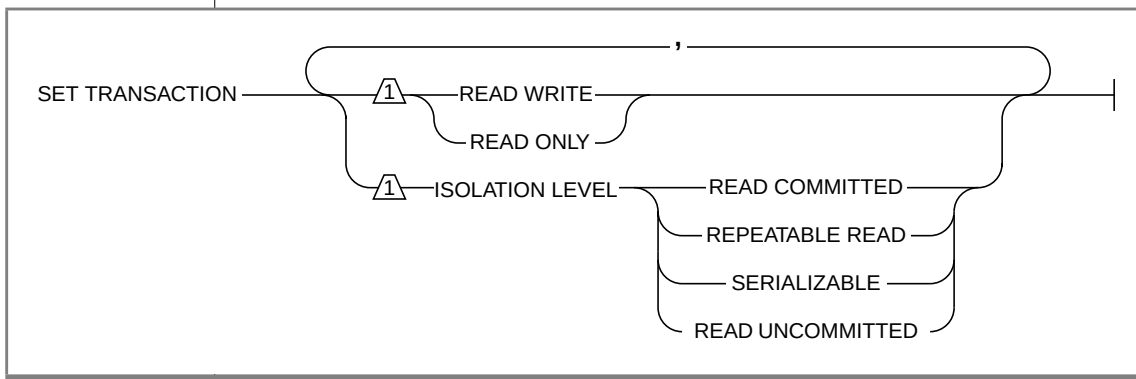
References

Related statements: CONNECT, DATABASE, GRANT, and SET ROLE

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. The database server uses shared locks to support different levels of isolation among processes that are attempting to read data as the following list shows:

- Read Uncommitted
- Read Committed
- (ANSI) Repeatable Read
- Serializable

The update or delete process always acquires an exclusive lock on the row that is being modified. The level of isolation does not interfere with 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 illustrate this difference in the behavior of the SET ISOLATION and SET TRANSACTION statements:

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;  
-- Produces error 876: Cannot issue SET TRANSACTION  
-- in an active transaction.
```

Another difference between SET ISOLATION and SET TRANSACTION is the duration of isolation levels. The isolation level set by SET ISOLATION remains in effect until another SET ISOLATION statement is issued. The isolation level set by SET TRANSACTION only remains in effect until the transaction terminates. Then the isolation level is reset to the default for the database type.

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 <i>phantom</i> 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 or insert a row that meets the search criteria of your query during your transaction. If you repeat the query during the transaction, you reread the same information. The shared locks are released only when the transaction commits or rolls back. Serializable is the default isolation level in an ANSI-compliant database.

In the following example, the user sets the isolation level to Serializable so that a query rereads the same information if it is repeated during the transaction:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

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

Informix database servers support 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 retrieval of BYTE or TEXT data can vary, depending on the database isolation levels. Under Read Uncommitted or Read Committed isolation levels, a process is permitted to read a BYTE or TEXT column that is either deleted (if the delete is not yet committed) or in the process of being deleted. Under these isolation levels, an application can read a deleted BYTE or TEXT column when certain conditions exist. For information about these conditions, see your [Administrator's Guide](#).

E/C

In ESQL/C, if you use a scroll cursor in a transaction, you can force consistency between your temporary table and the database table either by setting the isolation level to Serializable or by locking the entire table 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

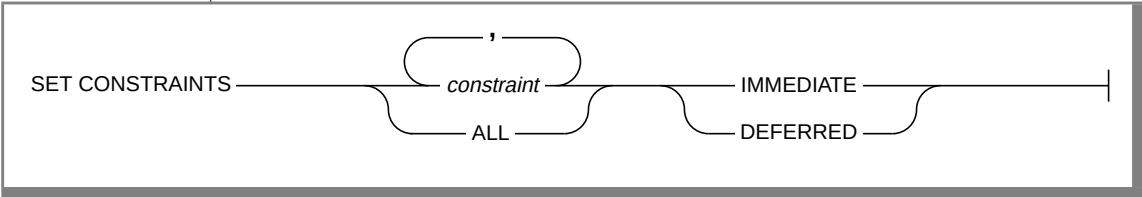
Related statements: CREATE DATABASE, SET ISOLATION, and SET LOCK MODE

For a discussion of isolation levels and concurrency issues, see the [Informix Guide to SQL: Tutorial](#).

SET Transaction Mode

Use the SET Transaction Mode statement to specify the transaction mode of constraints.

Syntax



Element	Purpose	Restrictions	Syntax
<i>constraint</i>	Constraint whose transaction mode is to be changed, or a list of constraint names No default value exists.	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.	Database Object Name, p. 4-25

Usage

To set the transaction mode of constraints, specify whether constraints are checked at the statement level or at the transaction level

When you set the transaction mode of a constraint, the effect of the SET Transaction Mode statement is limited to the transaction in which it is executed. The setting that the SET Transaction Mode statement produces is effective only during the transaction.

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, checking the not-null constraint for that column or set of columns is also deferred.*



Duration of Transaction Modes

The duration of the transaction mode that the SET Transaction Mode statement specifies is the transaction in which the SET Transaction Mode statement is executed. You cannot execute this statement outside a transaction. Once a COMMIT WORK or ROLLBACK WORK statement is successfully completed, the transaction mode of all constraints reverts to IMMEDIATE.

Switching Transaction Modes

To switch from transaction-level checking to statement-level checking, you can use the SET Transaction Mode statement to set the transaction mode to immediate, or you can use a COMMIT WORK or ROLLBACK WORK statement in your transaction.

Specifying All Constraints or a List of Constraints

You can specify all constraints in the database in your SET Transaction Mode statement, or you can specify a single constraint or list of constraints.

Specifying All Constraints

If you specify the ALL keyword, the SET Transaction Mode statement sets the transaction mode for all constraints in the database. If any statement in the transaction requires that any constraint on any table in the database be checked, the database server performs the checks at the statement level or the transaction level, depending on the setting that you specify in the SET Transaction Mode statement.

Specifying a List of Constraints

If you specify a single constraint name or a list of constraints, the SET Transaction Mode statement sets the transaction mode for the specified constraints only. If any statement in the transaction requires checking of a constraint that you did not specify in the SET Transaction Mode statement, that constraint is checked at the statement level regardless of the setting that you specified in the SET Transaction Mode statement for other constraints.

When you specify a list of constraints, the constraints do not have to be defined on the same table, but they must exist in the same database.

Specifying Remote Constraints

You can set the transaction mode of local constraints or remote constraints. That is, the constraints that are specified in the SET Transaction Mode statement can be constraints that are defined on local tables or constraints that are defined on remote tables.

Examples of Setting the Transaction Mode for Constraints

The following example shows how to defer checking constraints within a transaction until the transaction is complete. The SET Transaction Mode statement in the example specifies that any constraints on any tables in the database are not checked until the COMMIT WORK statement is encountered.

```
BEGIN WORK
SET CONSTRAINTS ALL DEFERRED
.
.
.
COMMIT WORK
```

The following example specifies that a list of constraints is not checked until the transaction is complete:

```
BEGIN WORK
SET CONSTRAINTS update_const, insert_const DEFERRED
.
.
.
COMMIT WORK
```

References

Related Statements: ALTER TABLE and CREATE TABLE

+

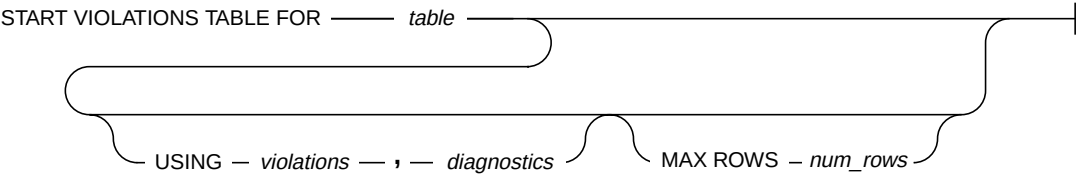
START VIOLATIONS TABLE

Use the START VIOLATIONS TABLE statement to create 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.

AD/XP

In Dynamic Server with AD and XP Options, the START VIOLATIONS TABLE statement creates a violations table but not a diagnostics table. Information that was stored in the diagnostics table has been incorporated into the violations table. ♦

Syntax



Element	Purpose	Restrictions	Syntax
<i>diagnostics</i>	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 .	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.	Database Object Name, p. 4-25

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>num_rows</i>	Maximum number of rows that can be inserted into the diagnostics table when a single statement (for example, INSERT or SET Database Object Mode) is executed on the target table	The value that you specify must be an integer in the range from 1 to the maximum value of the INTEGER data type. If you do not specify a value for <i>num_rows</i>, no upper limit exists on the number of rows that can be inserted into the diagnostics table when a single statement is executed on the target table.	Literal Number, p. 4-139
<i>table</i>	Name of the target table for which a violations table and diagnostics table are to be created	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.	Database Object Name, p. 4-25
<i>violations</i>	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.	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.	Database Object Name, p. 4-25

(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 Database Object Mode Statements

The START VIOLATIONS TABLE statement is closely related to the SET Database Object Mode statement. If you use the SET Database Object Mode statement to set the constraints or unique indexes defined on a table to the filtering database 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 Database Object Mode statement to set a disabled constraint or disabled unique index to the enabled or filtering database 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 database 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 Database Object Mode statement to set the database objects to the enabled or filtering database 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 more information, see the STOP VIOLATIONS TABLE statement on [page 2-613](#).

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 Database Object Mode 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.

(1 of 2)

Column Name	Type	Explanation
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 Database Object Mode statement
informix_recowner	CHAR(8)	This column identifies the user who issued the statement that created this bad row.

(2 of 2)

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_reowner 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.

(1 of 2)

Privilege	Condition
Update	The user has the Update privilege on the informix_tupleid, informix_optype, and informix_reowner 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.
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 Database Object Mode statement to set a disabled constraint or disabled unique index to the enabled or filtering mode, and a violations table and diagnostics table exist for the target table, you must have the Insert privilege on the violations and diagnostics tables.

If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the SET Database Object Mode statement provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the SET Database Object Mode statement.

- The grantor of the initial set of privileges on the violations table is the same as the grantor of the privileges on the target table. For example, if 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_reowner** 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_reowner** 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_reowner** 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_reowner**, 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 database object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in cascade mode (the default mode) causes the violations and diagnostics tables to be dropped also. If the target table is dropped in the restricted mode, the 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 database object mode of the constraint or unique index to the enabled or disabled database 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 BYTE or TEXT columns, BYTE or TEXT data in the violations table is created in the same blob space as the BYTE or TEXT data in the target table.

Example of a Violations Table

To start a violations and diagnostics table for the target table named **customer** in the demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR customer
```

Because 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
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.

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.
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.)

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 Database Object Mode statement to set a disabled constraint or disabled unique index to the enabled or filtering mode, and a violations table and diagnostics table exist for the target table, you must have the Insert privilege on the violations and diagnostics tables.

If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the SET Database Object Mode statement provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the SET Database Object Mode statement.

- The grantor of the initial set of privileges on the diagnostics table is the same as the grantor of the privileges on the target table. For example, if the user **jenny** 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_viol**s and a diagnostics table named **cust_subset_diags** for the **cust_subset** table, as follows:

```
START VIOLATIONS TABLE FOR cust_subset
  USING cust_subset_viol, 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 2-599](#).

The following issues concern the structure and use of the diagnostics table:

- The MAX ROWS clause of the START VIOLATIONS TABLE statement sets a limit on the number of rows that can be inserted into the diagnostics table when you execute a single statement, such as an INSERT or SET Database Object Mode statement, on the target table.
- The MAX ROWS clause limits the number of rows only for operations in which the table functions as a diagnostics table.
- When a table functions as a diagnostics table, it cannot have triggers or constraints defined on it.
- When a table functions as a diagnostics table, users can create indexes on the table, even though the existence of an index affects performance. You cannot set unique indexes on the diagnostics table to the filtering database object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in 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 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

Related statements: SET DATABASE OBJECT MODE and STOP VIOLATIONS TABLE

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

Use the STOP VIOLATIONS TABLE statement to drop the association between a target table and the special violations and diagnostics tables.

Syntax

STOP VIOLATIONS TABLE FOR _____ *table* _____

Element	Purpose	Restrictions	Syntax
<i>table</i>	Name of the target table whose association with the violations and diagnostics table is to be dropped No default value exists.	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.	Database Object Name, p. 4-25

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.

In Dynamic Server with AD and XP Options, the diagnostics table does not exist. Instead, the violations table incorporates the functions of the diagnostics table. The STOP VIOLATIONS TABLE statement drops the association between the target table and the violations table. ♦

AD/XP

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

Related statements: SET DATABASE OBJECT MODE and START VIOLATIONS TABLE

For a discussion of database object modes and violation detection, see the [*Informix Guide to SQL: Tutorial*](#).

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*](#).

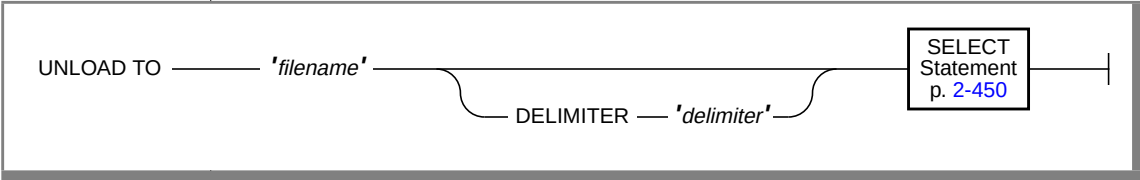
- +
- DB
- SQL

UNLOAD

Use the UNLOAD statement to write the rows retrieved in a SELECT statement to an operating-system file.

Use this statement with DB-Access and SQL Editor.

Syntax



Element	Purpose	Restrictions	Syntax
<i>delimiter</i>	Quoted string that identifies the character to use as the delimiter in 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 (\), newline character (=CTRL-J), hexadecimal numbers (0 to 9, a to f, A to F).	Quoted String, p. 4-157
<i>filename</i>	Quoted string that specifies the pathname and filename of an ASCII operating-system file The 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 that contains BYTE, TEXT, or VARCHAR data types to the output file, but you should be aware of the consequences. For further information, see “UNLOAD TO File” on page 2-616 .	Quoted String, p. 4-157 The pathname and filename must conform to the naming 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 2-342](#).

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 2-450](#).)

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
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.
date	DATE values are represented as <i>mm/dd/yyyy</i> , where <i>mm</i> is the month (January = 1, and so on), <i>dd</i> is the day, and <i>yyyy</i> 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.
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.

(1 of 2)

Data Type	Output Format
NULL	NULL columns are unloaded by placing no characters between the delimiters.
number	Number data types are displayed with no leading blanks. INTEGER or SMALLINT zero are represented as 0, and FLOAT, SMALLFLOAT, DECIMAL, or MONEY zero are represented as 0.00.
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: <i>yyyy-mm-dd hh:mi:ss.fff</i> , 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.

(2 of 2)

For more information on **DB** environment variables, refer to the [Informix Guide to SQL: Reference](#). For more information on **GL** environment variables, refer to the [Informix Guide to GLS Functionality](#).

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 [Informix Guide to GLS Functionality](#). ♦

Do not use the backslash (\) as a field separator or UNLOAD delimiter. It serves as an escape character to inform the UNLOAD command that the next character is to be interpreted as part of the data.

If you are unloading files that contain BYTE, TEXT, or VARCHAR data types, note the following information:

- BYTE items 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.
- Trailing blanks are retained in VARCHAR fields.
- Do not use the following characters as delimiters in the UNLOAD TO file: 0 to 9, a to f, A to F, newline character, or backslash.

GLS

If you are unloading files that contain data types, BYTE or TEXT columns 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. BYTE or TEXT columns 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](#).

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
```

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.

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 (\)
- Newline character (= CTRL-J)
- Hexadecimal numbers (0 to 9, a to f, A to F)

The following statement specifies the semicolon (;) as the delimiter character:

```
UNLOAD TO 'cust.out' DELIMITER ';'   
  SELECT fname, lname, company, city  
  FROM customer
```

References

Related statements: LOAD and SELECT

For information about setting the **DBDELIMITER** environment variable, see the [Informix Guide to SQL: Reference](#).

For a discussion of the GLS aspects of the UNLOAD statement, see the [Informix Guide to GLS Functionality](#).

For a task-oriented discussion of the UNLOAD statement and other utilities for moving data, see the [Informix Migration Guide](#).



UNLOCK TABLE

Use the `UNLOCK TABLE` statement in a database without transactions to unlock a table that you previously locked with the `LOCK TABLE` statement. The `UNLOCK TABLE` statement fails in a database that uses transactions.

Syntax

UNLOCK TABLE table
synonym

Element	Purpose	Restrictions	Syntax
<i>synonym</i>	Name of the synonym for the table you want to unlock	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table that you want to unlock	The table must be in database without transactions. The table must be one that you previously locked with the LOCK TABLE statement. You cannot unlock a table that another process locked.	Database Object Name, p. 4-25

Usage

You can lock a table if you own the table or if you have the `Select` privilege on the table, either from a direct grant to yourself or from a grant to **public**. You can only unlock a table that you locked. You cannot unlock a table that another process locked. Only one lock can apply to a table at a time.

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 following example shows how to change the lock mode of a table in a database that was created without transactions:

```
LOCK TABLE items IN EXCLUSIVE MODE
.  
.  
UNLOCK TABLE items  
.  
.  
LOCK TABLE items IN SHARE MODE
```

The UNLOCK TABLE statement fails if it is issued within a transaction. Table locks set within a transaction are released automatically when the transaction completes.

ANSI

If you are using an ANSI-compliant database, do not issue an UNLOCK TABLE statement. The UNLOCK TABLE statement fails if it is issued within a transaction, and a transaction is always in effect in an ANSI-compliant database. ♦

References

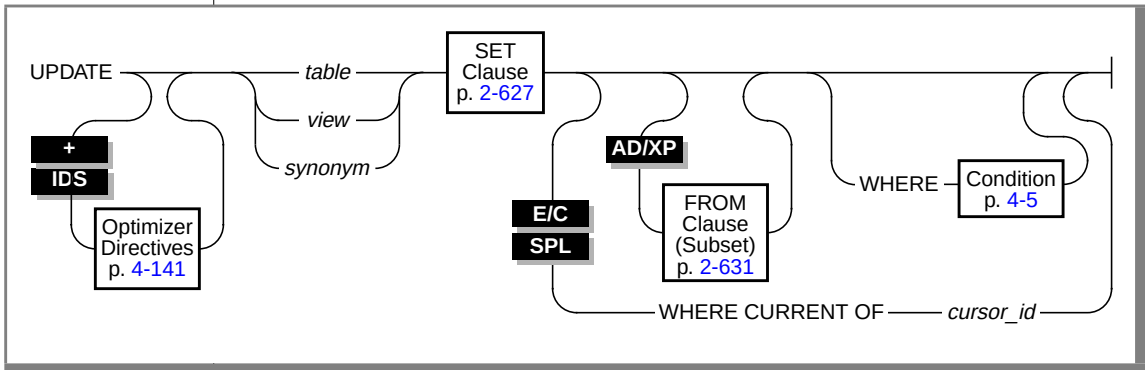
Related statements: BEGIN WORK, COMMIT WORK, LOCK TABLE, and ROLLBACK WORK

For a discussion of concurrency and locks, see the [Informix Guide to SQL: Tutorial](#).

UPDATE

Use the UPDATE statement to change the values in one or more columns of one or more rows in a table or view.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor_id</i>	Name of the cursor to use The current row of the active set for this cursor is updated when the UPDATE statement is executed. For more information on this parameter, see “WHERE CURRENT OF Clause” on page 2-632 .	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. 4-113
<i>synonym</i>	Name of the synonym that contains the rows to update	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table that contains the rows to update	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view that contains the rows to update	The view must exist.	Database Object Name, p. 4-25

Usage

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 2-342](#)). To update data in a view, you must have the Update privilege, and the view must meet the requirements that are explained in [“Updating Rows Through a View” on page 2-624](#).

If you omit the WHERE clause, all rows of the target table are updated.

AD/XP

DB

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.

In Dynamic Server with AD and XP Options, if the UPDATE statement is constructed in such a way that a single row might be updated more than once, the database server returns an error. However, if the new value is the same in every update, the database server allows the update operation to take place without reporting an error. ♦

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 2-342](#)). 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 more information, refer to the WITH CHECK OPTION discussion in the CREATE VIEW statement on [page 2-230](#).



Because duplicate rows can occur in a view even though the underlying table has unique rows, be careful when you update a table through a view. For example, if a view is defined on the **items** table and contains only the **order_num** and **total_price** columns, and if two items from the same order have the same total price, the view contains duplicate rows. In this case, if you update one of the two duplicate total price values, you have no way to know which item price is updated.

Important: *If you are using a view with a check option, you cannot update rows to a remote table.*

Updating Rows in a Database Without Transactions

If you are updating rows in a database without transactions, you must take explicit action to restore updated rows. For example, if the UPDATE statement fails after updating some rows, the successfully updated rows remain in the table. You cannot automatically recover from a failed update.

Updating Rows in a Database with Transactions

If you are updating rows in a database with transactions, and you are using transactions, you can undo the update using the ROLLBACK WORK statement. If you do not execute a BEGIN WORK statement before the update, and the update fails, the database server automatically rolls back any database modifications made since the beginning of the update.

You can create temporary tables with the WITH NO LOG option. These tables are never logged and are not recoverable.

AD/XP

In Dynamic Server with AD and XP Options, tables that you create with the RAW table type are never logged. Thus, RAW tables are not recoverable, even if the database uses logging. For information about RAW tables, refer to the [Informix Guide to SQL: Reference](#). ♦

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.

If you are within an explicit transaction, and the update fails, the database server automatically undoes the effects of the update. ♦

Locking Considerations

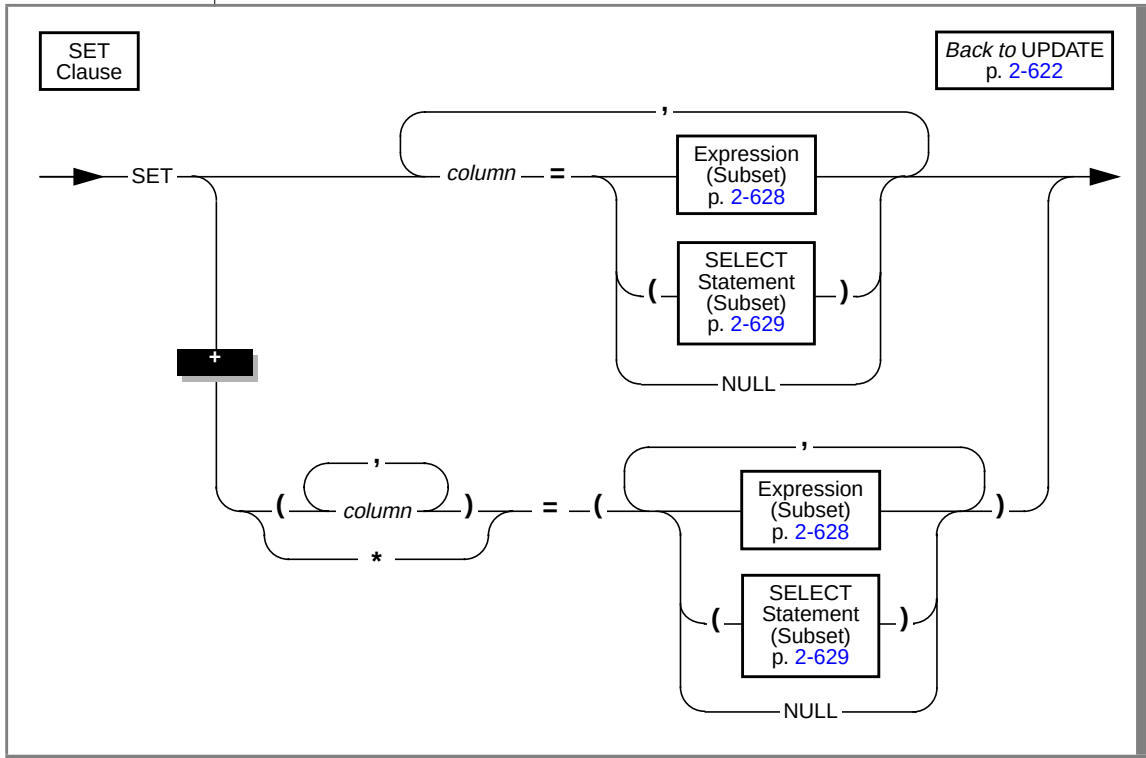
When a row is selected with the intent to update, the update process acquires an update lock. Update locks permit other processes to read, or *share*, a row that is about to be updated but do not let those processes update or delete it. Just before the update occurs, the update process *promotes* the 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.

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

The SET clause identifies the columns to be updated and assigns values to each column. The clause either pairs a single column to a single expression or lists multiple columns and sets them equal to corresponding expressions.



Element	Purpose	Restrictions	Syntax
*	Character that indicates all columns in the specified table or view are to be updated	The restrictions that apply to the “multiple columns equal to multiple expressions” format discussed under <i>column</i> also apply to the asterisk (*).	The asterisk (*) is a literal value with a special meaning in this statement.
<i>column</i>	Name of the column or columns that you want to update You can use either of two formats to specify multiple columns. These two formats are single columns to single expressions and multiple columns equal to multiple expressions. For further information on these formats, see “Single Columns to Single Expressions” on page 2-629 and “Multiple Columns Equal to Multiple Expressions” on page 2-630.	You cannot update SERIAL columns. If you use the format that pairs a single column to a single expression, you can include any number of single-column to single-expressions in the UPDATE statement. If you use the format that lists multiple columns and sets them equal to corresponding expressions, 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. 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. 4-113

Subset of Expressions Allowed in the SET Clause

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 4-33](#).

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 2-450](#).

Single Columns to Single Expressions

You can include any number of single-column to single-expressions in an 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
```

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
```

Multiple Columns Equal to Multiple Expressions

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 SET clause, you must list each expression explicitly, placing commas between expressions and enclosing the set of expressions in parentheses. 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 illustrate the multiple-column to multiple-expression form of the SET clause:

```
UPDATE customer
  SET (fname, lname) = ('John', 'Doe')
  WHERE customer_num = 101

UPDATE manufact
  SET * = ('HNT', 'Hunter')
  WHERE manu_code = 'ANZ'

UPDATE items
  SET (stock_num, manu_code, quantity) =
    ((SELECT stock_num, manu_code FROM stock
      WHERE description = 'baseball'), 2)
  WHERE item_num = 1 AND order_num = 1001

UPDATE table1
  SET (col1, col2, col3) =
    ((SELECT MIN (ship_charge),
      MAX (ship_charge) FROM orders),
     '07/01/1997')
  WHERE col4 = 1001
```

Updating the Same Column Twice

You can specify the same column more than once in the SET clause. If you do so, the column is set to the last value that you specified for the column. In the following example, the user specifies the **fname** column twice in the SET clause. For the row where the customer number is 101, the user sets **fname** first to gary and then to harry. After the UPDATE statement has been executed, the value of **fname** is harry.

```
UPDATE customer
  SET fname = "gary", fname = "harry"
  WHERE customer_num = 101
```

AD/XP

FROM Clause Subset

In Dynamic Server with AD and XP Options, you can use a join to determine the column values to update by supplying a FROM clause. You can use columns from any table that is listed in the FROM clause in the SET clause to provide values for the columns and rows to update.

You cannot use the LOCAL keyword or the SAMPLES OF segment of the FROM clause with the UPDATE statement.

For the full syntax of the FROM Clause, see [page 2-466](#).

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 2-450](#)). 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)
```

When you use the UPDATE statement with the WHERE clause, and no rows are updated, the SQLNOTFOUND value is 100 in ANSI-compliant databases and 0 in databases that are not ANSI compliant. If the UPDATE...WHERE...is a part of a multistatement prepare, and no rows are returned, the SQLNOTFOUND value is 100 for ANSI-compliant databases and databases that are not ANSI compliant.

WHERE CURRENT OF Clause

In ESQL/C, you can use the CURRENT OF keyword to update the current row of the active set of a cursor. However, you cannot update a row with a cursor if that row includes aggregates. The cursor named in the 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.

You can restrict the effect of the CURRENT OF keyword if you associate the UPDATE statement with a cursor that was created with the FOR UPDATE keyword. (See the DECLARE statement on [page 2-241](#).) If you created the cursor without specifying any columns for updating, you can update any column in a subsequent UPDATE...WHERE CURRENT OF statement. However, if the DECLARE statement that created the cursor specified one or more columns in the FOR UPDATE clause, you are restricted to updating only those columns in a subsequent UPDATE...WHERE CURRENT OF statement. The advantage to specifying columns in the FOR UPDATE clause of a DECLARE statement is speed. The database server can usually perform updates more quickly if columns are specified in the DECLARE statement. ♦

The following ESQL/C example illustrates the WHERE CURRENT OF form of the WHERE clause. In this example, updates are performed on a range of customers who receive 10-percent discounts (assume that a new column, **discount**, is added to the **customer** table). The UPDATE statement is prepared outside the WHILE loop to ensure that parsing is done only once. (For more information, see the PREPARE statement on [page 2-403](#).)

```
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;
}

```



Tip: You can use an update cursor to perform updates that are not possible with the UPDATE statement. An update cursor is a sequential cursor that is associated with a SELECT statement, which is declared with the FOR UPDATE keyword. For more information on the update cursor, see [page 2-245](#).

References

Related statements: DECLARE, INSERT, OPEN, and SELECT

For a task-oriented discussion of the UPDATE statement, see the [Informix Guide to SQL: Tutorial](#).

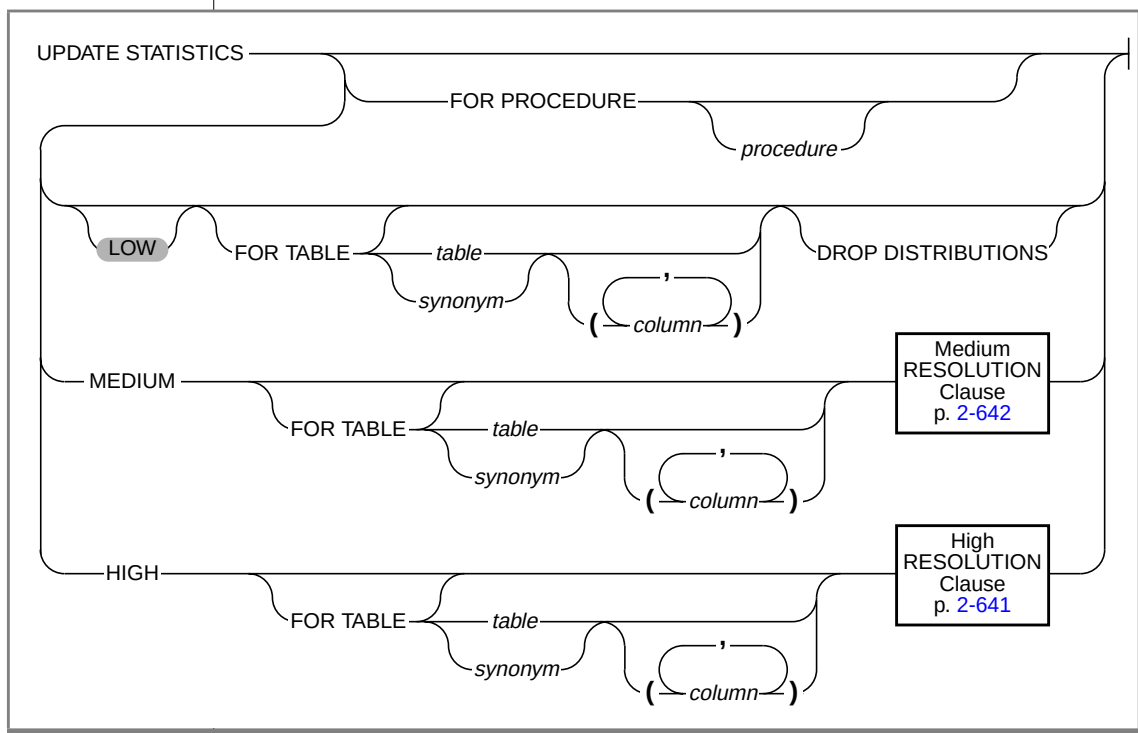
For a discussion of the GLS aspects of the UPDATE statement, see the [Informix Guide to GLS Functionality](#).

+

UPDATE STATISTICS

Use the UPDATE STATISTICS statement to update system catalog tables with information used to determine optimal query plans. In addition, you can use the UPDATE STATISTICS statement to force stored procedures to be reoptimized. If you upgrade to a new version of the database server, you can use UPDATE STATISTICS to convert table indexes to the format that the new database server uses.

Syntax



Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of a column in the specified table	The column must exist. If you use the LOW keyword and want the UPDATE STATISTICS statement to do minimal work, specify a column name that is not part of an index. If you use the MEDIUM or HIGH keywords, <i>column</i> cannot be a BYTE or TEXT column.	Identifier, p. 4-113
<i>procedure</i>	Name of procedure for which statistics are updated If you omit a procedure name after the FOR PROCEDURE keywords, the statistics for all stored procedures in the current database are updated.	The stored procedure must reside in the current database.	Database Object Name, p. 4-25
<i>synonym</i>	Name of synonym for table for which statistics are updated If you omit a table or synonym name after the FOR TABLE keywords, the statistics for all tables, including temporary tables, in the current database are updated.	The synonym and the table to which the synonym points must reside in the current database.	Database Object Name, p. 4-25
<i>table</i>	Name of table for which statistics are updated If you omit a table or synonym name after the FOR TABLE keywords, the statistics for all tables, including temporary tables, in the current database are updated.	The table must reside in the current database.	Database Object Name, p. 4-25

Usage

When you issue an UPDATE STATISTICS statement for a table, the database server recalculates the data in the **systables**, **syscolumns**, **sysindexes**, and **sysdistrib** system catalog tables. The optimizer uses this data to determine the best execution path for queries. The database server does not update this statistical data automatically. Statistics are updated only when you issue an UPDATE STATISTICS statement.

You can also issue an UPDATE STATISTICS statement to update the optimized execution plans for procedures in the **sysprocplan** system catalog table. Each time a procedure executes, the database server reoptimizes its execution plan if any objects that are referenced in the procedure have changed.

The UPDATE STATISTICS statement requires a current database. You cannot update the statistics used by the optimizer for a table or procedure that is external to the current database. This restriction applies to all modes of the UPDATE STATISTICS statement (LOW, MEDIUM, and HIGH).

If you omit the FOR TABLE or FOR PROCEDURE clauses, statistics are updated for every table and procedure in the current database, including the system catalog tables. Similarly, if you use the FOR TABLE or FOR PROCEDURE clauses and do not specify a table or procedure name, the database server updates the statistics for all tables, including temporary tables, or all procedures in the current database.

AD/XP

In Dynamic Server with AD and XP Options, the UPDATE STATISTICS statement does not update, maintain, or collect statistics on indexes. The statement does not update the **syscolumns** and **sysindexes** tables. In addition, the UPDATE STATISTICS statement does not reoptimize stored procedures. Any information about indexes or stored procedures in the following pages does not apply to Dynamic Server with AD and XP Options. ♦

Examining Index Pages

In Dynamic Server, the UPDATE STATISTICS statement directs the database server to read through index pages to compute statistics for the query optimizer. In addition, the server looks for pages where the delete flag is marked as 1. If pages are found with the delete flag marked as 1, the corresponding keys are removed from the B-tree cleaner list.

This operation is particularly useful if a system crash causes the B-tree cleaner list (which exists in shared memory) to be lost. To remove the B-tree items that have been marked as deleted but are not yet removed from the B-tree, run the UPDATE STATISTICS statement. For information on the B-tree cleaner list, see your [Administrator's Guide](#).

Updating Statistics When You Modify Tables

Update the statistics under the following circumstances:

- when you perform extensive modifications to a table
- when changes are made to tables that are used by one or more procedures, and you do not want the database server to reoptimize the procedure at execution time

If your application makes many modifications to the data in a particular table, update the system catalog table data for that table routinely with the UPDATE STATISTICS statement to improve query efficiency. The term “many modifications” is relative to the resolution of the distributions. In addition, if the data changes do not change the distribution of column values, you do not need to execute UPDATE STATISTICS again.

Updating Statistics When You Upgrade the Database Server

In Dynamic Server, when you upgrade a database to use with a newer database server, you can use the UPDATE STATISTICS statement to convert the indexes to the form that the newer database server uses. You can choose to convert the indexes one table at a time or for the entire database at one time. Follow the conversion guidelines that are outlined in the [Informix Migration Guide](#). When you use the UPDATE STATISTICS statement to convert the indexes to use with a newer database server, the indexes are implicitly dropped and re-created. The only time that an UPDATE STATISTICS statement causes table indexes to be implicitly dropped and recreated is when you upgrade a database for use with a newer database server.

Modes of UPDATE STATISTICS

You can specify three modes in which UPDATE STATISTICS is run: low, medium, and high. You use the LOW, MEDIUM, and HIGH keywords to specify these modes. The following table shows the different modes of UPDATE STATISTICS and the purpose of each mode.

Mode	Purpose
LOW	The database server updates the statistical data in the systables , syscolumns , and sysindexes system catalog tables. This mode causes the least amount of information to be gathered. This mode executes more quickly than the HIGH and MEDIUM modes.
HIGH	In addition to performing the functions of the LOW mode, the database server updates the data distribution statistics in the sysdistrib system catalog table. The constructed distribution is exact. Because of the time required to gather this information, this mode executes more slowly than the LOW and MEDIUM modes.
MEDIUM	In addition to performing the functions of the LOW mode, the database server updates the data distribution statistics in the sysdistrib system catalog table. The database server obtains the data for the distributions by sampling rather than by scanning all the rows. This mode executes more quickly than the HIGH mode but more slowly than the LOW mode.

Specifying the LOW Keyword

If you use the **LOW** keyword, or if you specify no keyword, the database server collects the smallest amount of information about the column. The database server updates table, row, and page counts as well as index and column statistics for specified columns. The data in the **systables**, **syscolumns**, and **sysindexes** tables is updated.

When you use the **LOW** mode, no information is put into the **sysdistrib** system catalog table. If distribution data already exists in the **sysdistrib** system catalog table, the data remains intact unless you use the **DROP DISTRIBUTIONS** option. If you specify the **DROP DISTRIBUTIONS** option but do not specify a table name, all the distribution information is removed.

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)
```

If you want the **UPDATE STATISTICS** statement to do minimal work, specify a column that is not part of an index.

Dropping Data with the DROP DISTRIBUTIONS Clause

If you want to drop distribution data for some or all of the columns that are already defined in the **sysdistrib** table, but you want to update the statistics with the **LOW** option for the rest of the columns in the table, you can use the **DROP DISTRIBUTIONS** clause. If you specify the **DROP DISTRIBUTIONS** keyword, all distribution information that exists for the column specified in the **UPDATE STATISTICS** statement drops. If no columns are specified, all of the distributions for that table are removed.

You must have the **DBA** privilege or be the owner of the table 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
```

Creating Distributions for Columns

When you run UPDATE STATISTICS in either the HIGH or MEDIUM mode, the database server examines the contents of the specified columns and divides them into bins, which represent a percentage of data. For example, a bin might hold 2 percent of the data; 50 bins would hold all of the data. You can set the size of the bin with the RESOLUTION *percent* parameter.

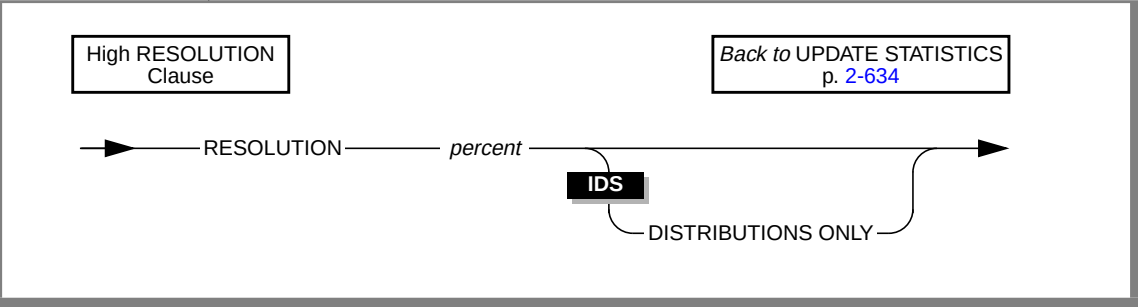
The organization of column values into bins is called the distribution (for that column). The optimizer examines distributions of columns that are referenced in a WHERE clause to estimate the cost effect of a WHERE clause on the query.

You can use the MEDIUM or HIGH keywords to specify the mode for data distributions on specific columns. These keywords indicate that the database server is to generate statistics about the distribution of data values for each specified column and place that information in a system catalog table called **sysdistrib**.

You cannot create distributions for BYTE or TEXT columns. If you include a BYTE or TEXT column in an UPDATE STATISTICS statement that specifies medium or high distributions, no distributions are created for those columns. Distributions are constructed for other columns in the list, and the statement does not return an error.

You must have the DBA privilege or be the owner of the table to create high or medium distributions.

High RESOLUTION Clause



Element	Purpose	Restrictions	Syntax
<i>percent</i>	Desired resolution in units of percent, so that 0.1 means the data in a column is divided into bins, each containing (on average) 0.1 percent of the data	The minimum resolution possible for a table is 1 / <i>nrows</i> , where <i>nrows</i> is the number of rows in the table.	Literal Number, p. 4-139
The default value is 0.5.			

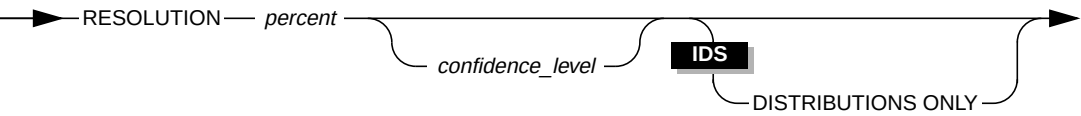
If you use the HIGH keyword, the constructed distribution is exact, rather than statistically significant. Because of the time required to gather the information, you should use high distributions for specific tables or even columns rather than for the entire database. For large tables, the database server might scan the data once for each column. The amount of space that the **DBUPSPACE** environment variable designates determines the number of times the table is scanned. For information about **DBUPSPACE**, see the [Informix Guide to SQL: Reference](#).

If you do not specify a RESOLUTION clause, the default percentage is 0.5.

Medium RESOLUTION Clause

Medium RESOLUTION
Clause

Back to UPDATE STATISTICS
p. 2-634



Element	Purpose	Restrictions	Syntax
<i>confidence_level</i>	Expected fraction of times that the sampling entailed by the MEDIUM keyword should produce the same results as the exact methods entailed by the HIGH keyword The default confidence level is 0.95. This can be roughly interpreted as meaning that 95 percent of the time, the estimate produced by the MEDIUM keyword is equivalent to using high distributions.	The minimum confidence level is 0.80. The maximum confidence level is 0.99.	Literal Number, p. 4-139
<i>percent</i>	Desired resolution in units of percent, so that 0.1 means the data in a column is divided into bins, each containing (on average) 0.1 percent of the data The default value is 2.5.	The minimum resolution possible for a table is 1 / <i>nrows</i> , where <i>nrows</i> is the number of rows in the table.	Literal Number, p. 4-139

If you use the MEDIUM keyword, the data for the distributions is obtained by sampling. Because the data obtained by sampling is usually much smaller than the actual number of rows, the time required to construct medium distributions is less than the time required for high mode. Medium distributions require at least one scan of the table, so the creation of medium distributions executes more slowly than the creation of low distributions.

If you do not specify a `RESOLUTION` clause, the default percentage is 2.5. If you do not specify a value for `confidence_level`, the default confidence is 0.95. This value can be roughly interpreted to mean that 95 percent of the time, the estimate is equivalent to that obtained from high distributions.

Specifying DISTRIBUTIONS ONLY to Suppress Index Information

In Dynamic Server, when you specify the `MEDIUM` or `HIGH` keywords, your `UPDATE STATISTICS` statement performs the functions of the `LOW` keyword as well. The `LOW` keyword constructs table information and index information for the specified objects. If you specify the `DISTRIBUTIONS ONLY` option with the `MEDIUM` or `HIGH` keywords, you leave the existing index information in place. However, table and column information is still constructed for the specified objects when you specify the `DISTRIBUTIONS ONLY` option. This information includes the number of pages used, the number of rows, and fragment information.

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 leaves the existing index information in place.

```
UPDATE STATISTICS MEDIUM FOR TABLE customer
DISTRIBUTIONS ONLY
```

UPDATE STATISTICS and Temporary Tables

You can use `UPDATE STATISTICS` on temporary tables. Specify the name of the table to explicitly update the statistics for a temporary table or build distributions for a temporary table. If you use the `FOR TABLE` clause without a specific table name to build distributions on all of the tables in the database, distributions will also be built on all of the temporary tables in your session.

References

Related statements: SET EXPLAIN and SET OPTIMIZATION

For a discussion of the performance implications of UPDATE STATISTICS, see your [Performance Guide](#).

For a discussion of how to use the **dbschema** utility to view distributions created with UPDATE STATISTICS, see the [Informix Migration Guide](#).

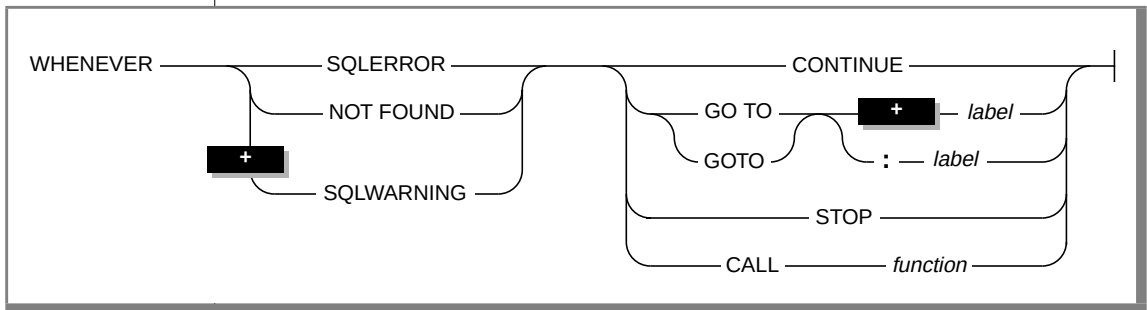
E/C

WHENEVER

Use the WHENEVER statement to trap exceptions that occur during the execution of SQL statements.

Use this statement with ESQL / C.

Syntax



Element	Purpose	Restrictions	Syntax
<i>function</i>	Name of the function or procedure that is called when an exception occurs	Function or procedure must exist at compile time.	Name must conform to language-specific rules for functions or procedures.
<i>label</i>	Statement label to which program control transfers when an exception occurs	A label must be a paragraph name or a procedure name.	Label must conform to language-specific rules for statement labels.

Usage

The WHENEVER statement is equivalent to placing an exception-checking routine after every SQL statement. The following table summarizes the types of exceptions for which you can check with the WHENEVER statement.

Type of Exception	WHENEVER Clause	For More Information
Errors	SQLERROR	page 2-648
Warnings	SQLWARNING	page 2-648
Not Found Condition End of Data Condition	NOT FOUND	page 2-649

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 2-649
Stop program execution	STOP	page 2-649
Transfer control to a specified label	GOTO GO TO	page 2-650
Transfer control to a named function or procedure	CALL	page 2-651

The Scope of WHENEVER

The ESQL/C preprocessor, not the database server, handles the interpretation of the WHENEVER statement. When the preprocessor encounters a WHENEVER statement in an ESQL/C source file, it inserts 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
4  EXEC SQL connect to 'test';
5  EXEC SQL WHENEVER SQLERROR STOP;
6
7  EXEC SQL WHENEVER NOT FOUND CONTINUE;
8
9  printf("\n\nGoing to try first insert\n\n");
10
11
12

```

```
6 EXEC SQL insert into test_color values ('green');

7 EXEC SQL WHENEVER NOT FOUND CONTINUE;
8 EXEC SQL WHENEVER SQLERROR CONTINUE;

9 printf("\n\nGoing to try second insert\n\n");
10 EXEC SQL insert into test_color values ('blue');
11 EXEC SQL select paint_type from paint where color='red';
12 EXEC SQL disconnect all;
13 printf("\n\nProgram over\n\n");
14 }
```

SQLERROR Keyword

If you use the SQLERROR keyword, any SQL statement that encounters an error is handled as the WHENEVER SQLERROR statement directs. If an error occurs, the **sqlcode** variable (sqlca.sqlcode, SQLCODE) is 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.

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 is set to 100, and the SQLSTATE variable has a class code of 02. For the name of the **sqlcode** variable in each Informix product, see the table in [“SQLERROR Keyword” on page 2-648](#).

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 Keywords

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 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.
- You cannot specify the name of a stored procedure as a named function. To call a stored procedure, use the CALL clause to execute a function that contains the EXECUTE PROCEDURE statement.
- Make sure that all functions that the WHENEVER...CALL statement affects can find a declaration of the named function.

References

Related Statements: EXECUTE PROCEDURE and FETCH

For discussions of exception checking and error checking, see the [*INFORMIX-ESQL/C Programmer's Manual*](#).

SPL Statements

CALL	3-4
CONTINUE	3-7
DEFINE	3-8
EXIT	3-16
FOR.	3-18
FOREACH	3-23
IF	3-27
LET	3-31
ON EXCEPTION	3-34
RAISE EXCEPTION	3-39
RETURN	3-41
SYSTEM	3-44
TRACE	3-47
WHILE	3-50

Y

ou can use SQL statements and Stored Procedure Language (SPL) statements to write procedures, and you can store these procedures in the database. These stored procedures 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

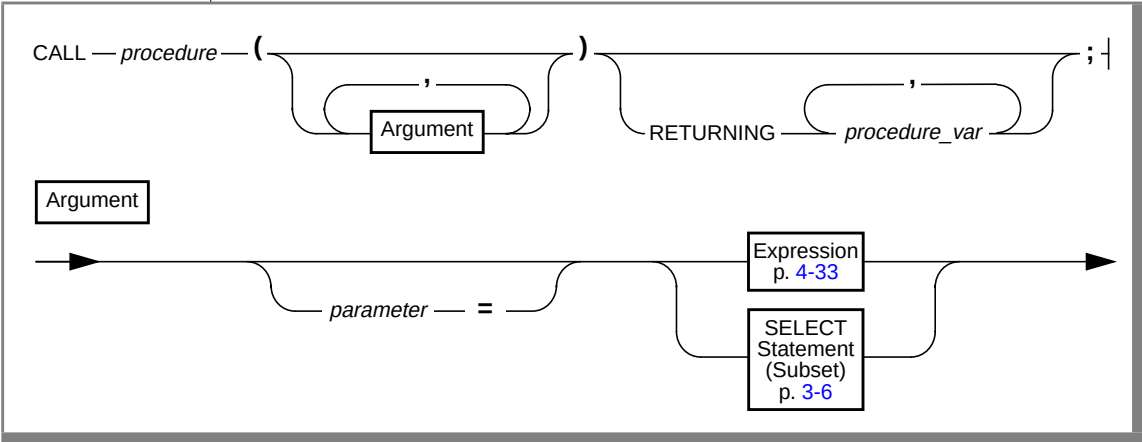
If a statement is composed of multiple clauses, the statement description provides the same set of information for each clause.

For task-oriented information about using stored procedures, see the [Informix Guide to SQL: Tutorial](#).

CALL

Use the CALL statement to execute a procedure from within a stored procedure.

Syntax



Element	Purpose	Restrictions	Syntax
<i>parameter</i>	Name of the parameter in the called procedure, as defined by its CREATE PROCEDURE statement	Name or position, but not both, binds procedure arguments to procedure parameters. That is, you can use the <i>parameter</i> = syntax for none or all the arguments that are specified in one CALL statement.	Identifier, p. 4-113
<i>procedure</i>	Name of the procedure to be called	The procedure must exist.	Database Object Name, p. 4-25
<i>procedure_var</i>	Name of a variable that receives the value being returned	The data type of <i>procedure_var</i> must match that of the value that is being returned.	Identifier, p. 4-113

Usage

The CALL statement invokes a procedure. The CALL statement is identical in behavior to the EXECUTE PROCEDURE statement, but you can only use it from within a stored procedure.

Specifying Arguments

If CALL statement contains more arguments than the called procedure expects, you receive an error.

If a CALL statement specifies fewer arguments than the called procedure expects, the arguments are said to be missing. The database server initializes missing arguments to their corresponding default values. (See CREATE PROCEDURE on [page 2-134](#).) This initialization occurs before the first executable statement in the body of the procedure.

If missing arguments do not have default values, the database server initializes the arguments to the value of UNDEFINED. An attempt to use any variable that has the value of UNDEFINED results in an error.

Either name or position, but not both, binds procedure arguments to procedure parameters. That is, you can use the *parameter* = syntax for all or none of the arguments that are specified in one CALL statement.

Each procedure call in the following example is valid for a procedure that expects character arguments t, n, and d, in that order:

```
CALL add_col (t='customer', n = 'newint', d ='integer');  
CALL  add_col('customer','newint','integer');
```


Subset of SELECT Allowed in a Procedure Argument

You can use any SELECT statement as the argument for a procedure if it returns exactly one value of the proper data type and length. For more information, see the discussion of SELECT statements on [page 2-450](#).

Receiving Input from the Called Procedure

The RETURNING clause specifies the procedure variable that receives the returned values from a procedure call. If you omit the RETURNING clause, the called procedure does not return any values.

The following example shows two procedure calls, one that expects no returned values (**no_args**) and one that expects three returned values (**yes_args**). The creator of the procedure has defined three integer variables to receive the returned values from **yes_args**.

```
CREATE PROCEDURE not_much()  
  DEFINE i, j, k INT;  
  CALL no_args (10,20);  
  CALL yes_args (5) RETURNING i, j, k;  
END PROCEDURE
```

CONTINUE

Use the CONTINUE statement to start the next iteration of the innermost FOR, WHILE, or FOREACH loop.

Syntax

```

CONTINUE _____
                  |
                  |____ FOR _____
                  |____ WHILE _____
                  |____ FOREACH _____
                  |_____ ; _____
  
```

Usage

When you encounter a CONTINUE statement, the procedure 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 procedure inserts values 3 through 15 into the table **testtable**. The procedure also returns values 3 through 9 and 13 through 15 in the process. The procedure does not return the value 11 because it encounters the CONTINUE FOR statement. The CONTINUE FOR statement causes the procedure to skip the RETURN I WITH RESUME statement.

```

CREATE PROCEDURE 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;

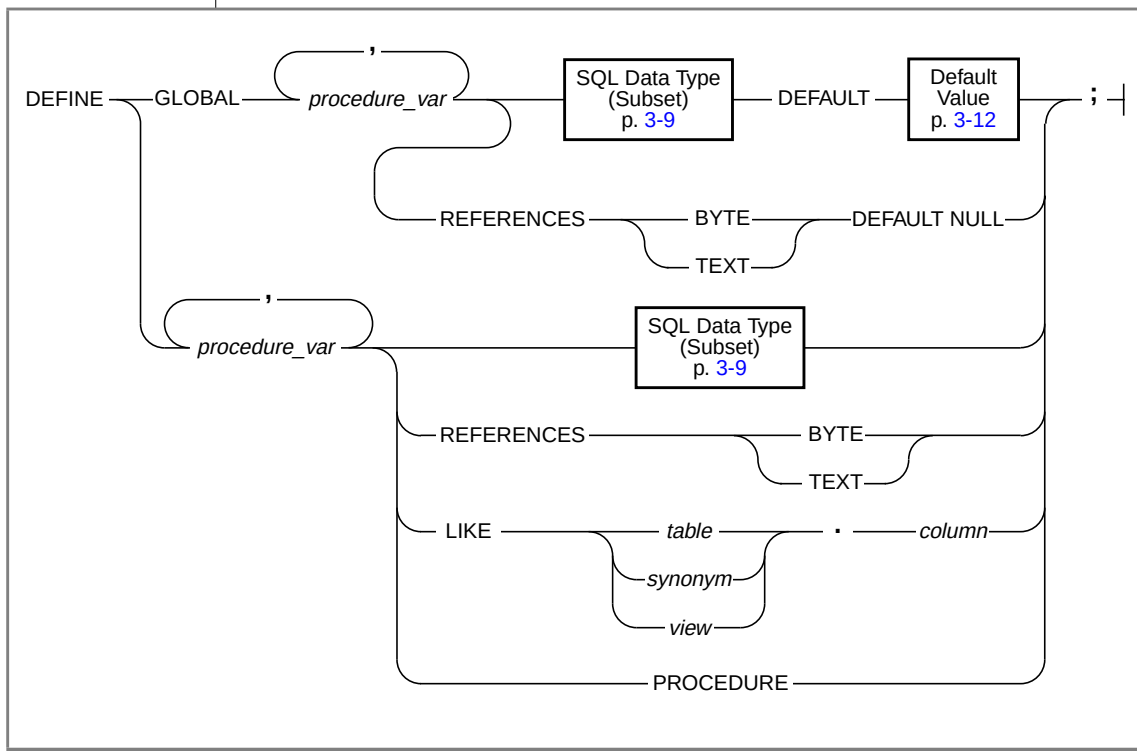
END PROCEDURE;
  
```

The CONTINUE statement generates errors if it cannot find the identified loop.

DEFINE

Use the DEFINE statement to declare variables that the procedure uses and to assign them data types.

Syntax



Element	Purpose	Restrictions	Syntax
<i>column</i>	Name of a column in the table	The column must already exist in the table.	Identifier, p. 4-113
<i>table</i>	Name of the table that contains the data type you want the new variable to match	The table must exist.	Database Object Name, p. 4-25
<i>synonym</i>	Name of the synonym that contains the data type you want the new variable to match	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view that contains the data type you want the new variable to match	The view must exist.	Database Object Name, p. 4-25
<i>procedure_var</i>	Name of the procedure variable that is being defined	The name must be unique to the statement block.	Identifier, p. 4-113

Usage

The DEFINE statement is not an executable statement. The DEFINE statement must appear after the procedure header and before any other statements. You can use a variable anywhere within the statement block where it is defined; that is, the scope of a defined variable is the statement block in which it was defined.

SQL Data Type Subset

The SQL data type subset includes all the SQL data types except BYTE, SERIAL, and TEXT.

Defining BYTE and TEXT Variables

The REFERENCES keyword lets you use BYTE and TEXT variables. BYTE and TEXT variables do not contain the actual data but are simply pointers to the data. The REFERENCES keyword is a reminder that the procedure variable is just a pointer. Use the procedure variables for BYTE and TEXT data types exactly as you would any other variable.

Redeclaration or Redefinition

If you define the same variable twice within the same statement block, you receive an error. You can redefine a variable within a nested block, in which case it temporarily hides the outer declaration. The following example produces an error:

```
CREATE PROCEDURE example1()
  DEFINE n INT; DEFINE j INT;
  DEFINE n CHAR (1); -- redefinition produces an error
  .
  .
  .
```

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
```

Declaring GLOBAL Variables

The GLOBAL modifier indicates that the list of variables that follows the GLOBAL keyword are available to other procedures. The data types of these variables must match the data types of variables in the *global environment*. The global environment is the memory that is used by all the procedures that run within a given session (a DB-Access session or an ESQL/C session). The values of global variables are stored in memory.

Procedures 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.

Databases do not share global variables. The database server and any application development tools do not share global variables.

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 procedures, **proc1** and **proc2**; each has defined the global variable **gl_out**:

Stored Procedure proc1

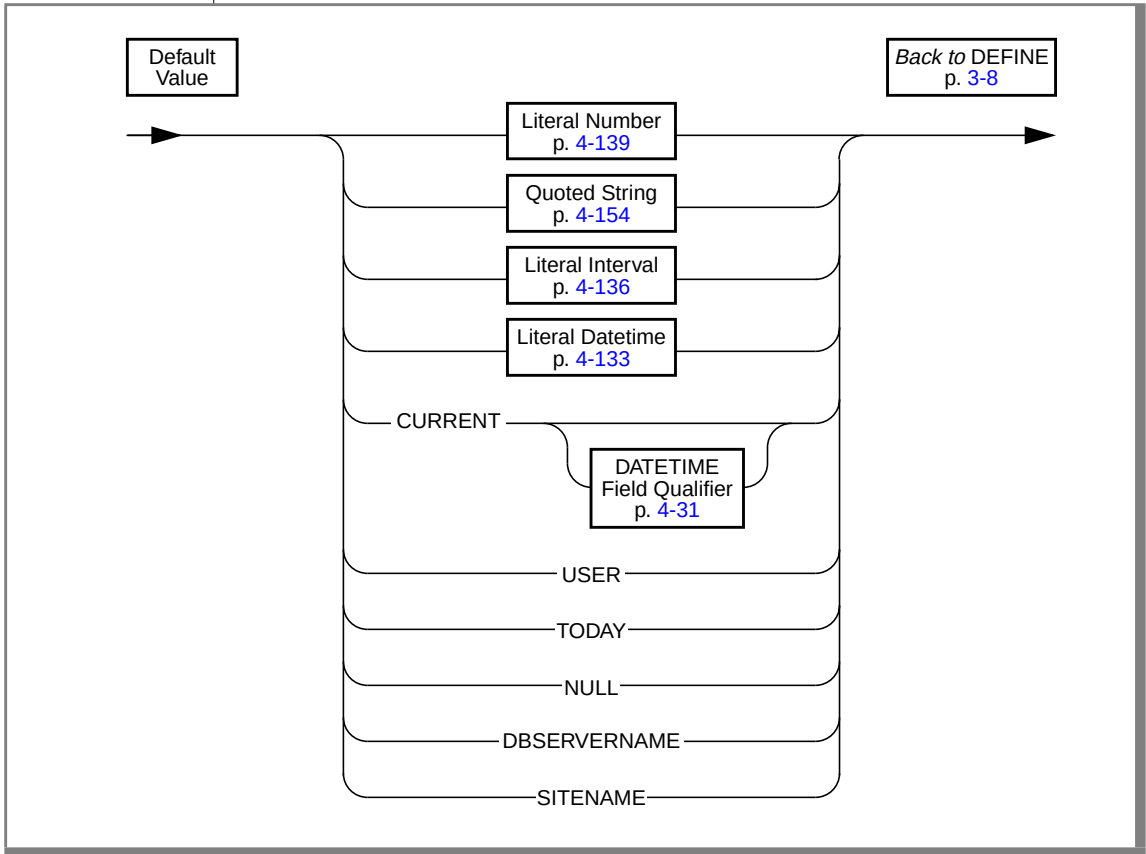
```
CREATE PROCEDURE proc1()
.
.
.
  DEFINE GLOBAL gl_out INT DEFAULT 13;
.
.
  LET gl_out = gl_out + 1;
END PROCEDURE;
```

Stored Procedure proc2

```
CREATE PROCEDURE proc2()
.
.
.
  DEFINE GLOBAL gl_out INT DEFAULT 23;
  DEFINE tmp INT;
.
.
.
  LET tmp = gl_out
.
.
.
END PROCEDURE;
```

If **proc1** is called first, **gl_out** is set to 13 and then incremented to 14. If **proc2** is then called, it sees that 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.

Default Value Clause



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.

BYTE and TEXT

The only default value possible for a BYTE or TEXT variable is null. The following example defines a TEXT global variable that is called **l_text**:

```
CREATE PROCEDURE use_text()
    DEFINE i INT;
    DEFINE GLOBAL l_text REFERENCES TEXT DEFAULT NULL;
END PROCEDURE
```


Declaring Local Variables

Nonglobal (local) 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 Variables LIKE Columns

If you use the LIKE clause, the database server defines *procedure_var* as the same data type as the *column* in *table*. The data types of variables that are defined as database columns are resolved at run time; therefore, *column* and *table* do not need to exist at compile time.

Declaring a Variable LIKE a SERIAL Column

You can use the LIKE keyword to declare that a variable is LIKE a SERIAL column. For example, if the column **serialcol** in the **mytab** table has the SERIAL data type, you can create the following procedure:

```
CREATE PROCEDURE proc1()
  DEFINE local_var LIKE mytab.serialcol;
  RETURN;
END PROCEDURE;
```

The variable **local_var** is treated as an INTEGER variable.

However, you cannot define a variable as a SERIAL variable. The following example generates an error:

```
CREATE PROCEDURE proc2()
  DEFINE local_var SERIAL;
  RETURN;
END PROCEDURE;
```

Declaring Variables as the PROCEDURE Type

The PROCEDURE type indicates that in the current scope, *procedure_var* is a user-defined procedure call and not an SQL function or a system function call. For example, the following statement defines **length** as a stored procedure, not as the SQL LENGTH function. This definition disables the SQL LENGTH function within the scope of the statement block. You would use such a definition if you had created a procedure with the name **length** before you defined and used it in another procedure, as shown in the following example:

```
DEFINE length PROCEDURE;
.
.
.
LET x = length (a,b,c)
```

If you create a procedure with the same name as an aggregate function (SUM, MAX, MIN, AVG, COUNT) or with the name **extend**, you must qualify the procedure name with the owner name.

Declaring Variables for BYTE and TEXT Data

The keyword REFERENCES indicates that *procedure_var* is not a BYTE or TEXT value but a pointer to the BYTE or TEXT value. Use the variable as though it holds the data.

The following example defines a local BYTE variable:

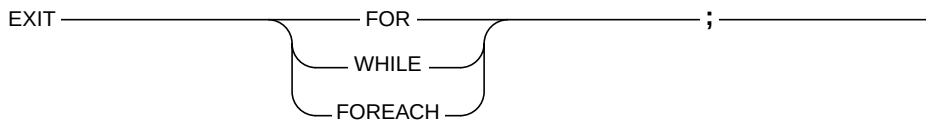
```
CREATE PROCEDURE use_byte()
  DEFINE i INT;
  DEFINE l_byte REFERENCES BYTE;
END PROCEDURE --use_byte
```

If you pass a variable of BYTE or TEXT data type to a procedure, 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 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.

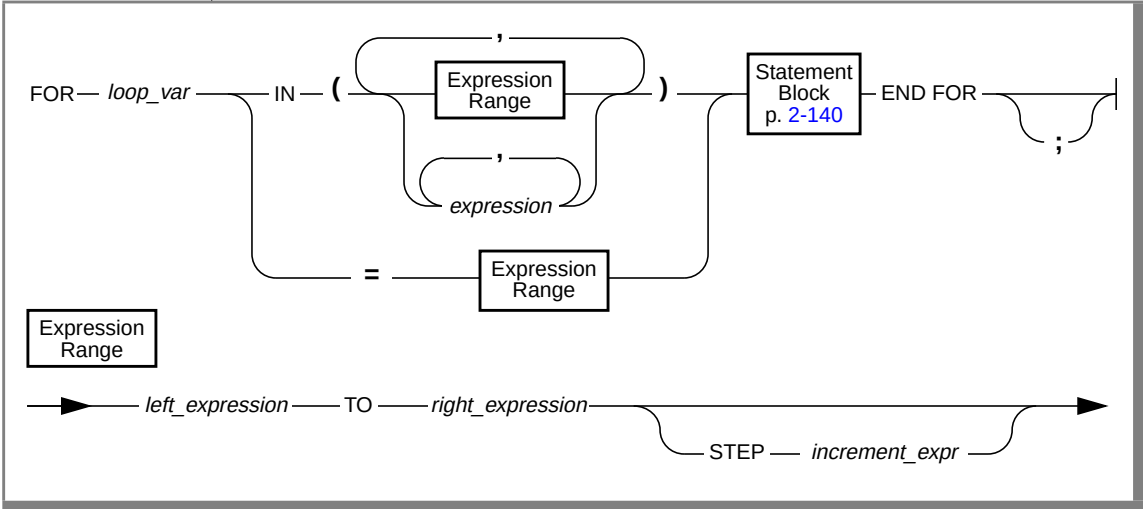
The following example uses an EXIT FOR statement. In the FOR loop, when *j* becomes 6, the IF condition *i* = 5 in the WHILE loop is true. The FOR loop stops executing, and the procedure continues at the next statement outside the FOR loop (in this case, the END PROCEDURE statement). In this example, 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
<i>expression</i>	Numeric or character value against which <i>loop_var</i> is compared to determine if the loop should be executed	The data type of <i>expression</i> must match the data type of <i>loop_var</i> . You can use the output of a SELECT statement as an <i>expression</i> .	Expression, p. 4-33
<i>increment_expr</i>	Positive or negative value by which <i>loop_var</i> is incremented. Defaults to +1 or -1 depending on <i>left_expression</i> and <i>right_expression</i> .	The increment expression cannot evaluate to 0.	Expression, p. 4-33

Element	Purpose	Restrictions	Syntax
<i>left_expression</i>	Starting expression of a range	The value of <i>left_expression</i> must match the data type of <i>loop_var</i> . It must be either INT or SMALLINT.	Expression, p. 4-33
<i>right_expression</i>	Ending expression in the range. The size of <i>right_expression</i> relative to <i>left_expression</i> determines if the range is stepped through positively or negatively.	The value of <i>right_expression</i> must match the data type of <i>loop_var</i> . It must be either INT or SMALLINT.	Expression, p. 4-33
<i>loop_var</i>	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 <i>loop_var</i> with a range of values and the TO keyword, you must define <i>loop_var</i> explicitly as either INT or SMALLINT.	Identifier, p. 4-113

(2 of 2)

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 *loop_var* 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 *loop_var*.

Using the TO Keyword to Define a Range

The TO keyword implies a range operator. The range is defined by *left_expression* and *right_expression*, and the STEP *increment_expr* option implicitly sets the number of increments. If you use the TO keyword, *loop_var* must be an INT or SMALLINT data type. The following example shows two equivalent FOR statements. Each uses the TO keyword to define a range. The first statement uses the IN keyword, and the second statement uses an equal sign (=). Each statement causes the loop to execute five times.

```
FOR index_var IN (12 TO 21 STEP 2)
  -- statement block
END FOR

FOR index_var = 12 TO 21 STEP 2
  -- statement block
END FOR
```

If you omit the STEP option, the database server gives *increment_expr* the value of -1 if *right_expression* is less than *left_expression*, or +1 if *right_expression* is more than *left_expression*. If *increment_expr* is specified, it must be negative if *right_expression* is less than *left_expression*, or positive if *right_expression* is more than *left_expression*. The two statements in the following example are equivalent. In the first statement, the STEP increment is explicit. In the second statement, the STEP increment is implicitly 1.

```
FOR index IN (12 TO 21 STEP 1)
  -- statement block
END FOR

FOR index = 12 TO 21
  -- statement block
END FOR
```

The database server initializes the value of *loop_var* to the value of *left_expression*. In subsequent iterations, the server adds *increment_expr* to the value of *loop_var* and checks *increment_expr* to determine whether the value of *loop_var* is still between *left_expression* and *right_expression*. If so, the next iteration occurs. Otherwise, an exit from the loop takes place. Or, if you specify another range, the variable takes on the value of the first element in the next range.

Specifying Two or More Ranges in a Single FOR Statement

The following example shows a statement that traverses a loop forward and backward and uses different increment values for each direction:

```
FOR index_var IN (15 to 21 STEP 2, 21 to 15 STEP -3)
  -- statement body
END FOR
```

Using an Expression List as the Range

The database server initializes the value of *loop_var* to the value of the first expression specified. In subsequent iterations, *loop_var* takes on the value of the next expression. When the server has evaluated the last expression in the list and used it, the loop stops.

The expressions in the IN list do not have to be numeric values, as long as you do not use range operators in the IN list. The following example uses a character expression list:

```
FOR c IN ('hello', (SELECT name FROM t), 'world', v1, v2)
  INSERT INTO t VALUES (c);
END FOR
```

The following FOR statement shows the use of a numeric expression list:

```
FOR index IN (15,16,17,18,19,20,21)
  -- statement block
END FOR
```


Mixing Range and Expression Lists in the Same FOR Statement

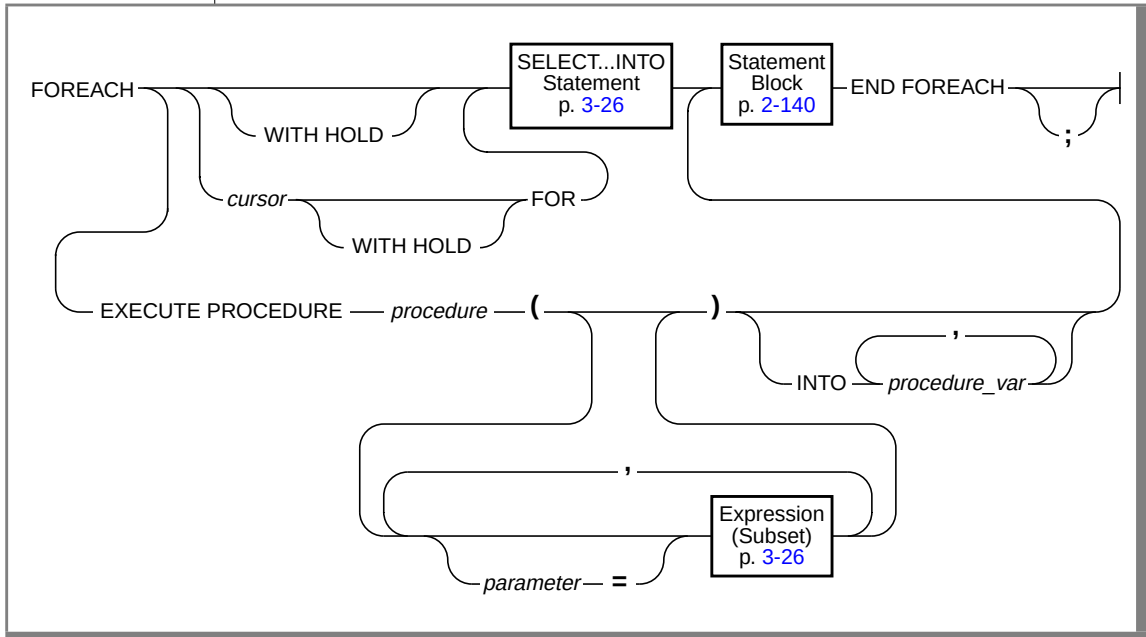
If *loop_var* is an INT or SMALLINT value, you can mix ranges and expression lists in the same FOR statement. The following example shows a mixture that uses an integer variable. Values in the expression list include the value that is returned from a SELECT statement, a sum of an integer variable and a constant, the values that are returned from a 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 manipulate more than one row.

Syntax



Element	Purpose	Restrictions	Syntax
<i>cursor</i>	Identifier that you supply as a name for the SELECT...INTO statement	Each cursor name within a procedure must be unique.	Identifier, p. 4-113
<i>parameter</i>	Name of a parameter in the procedure that is being executed as defined in its CREATE PROCEDURE statement	Name or position, but not both, binds procedure arguments to procedure parameters. You can use <i>parameter</i> = syntax for all or none of the arguments that are specified in one FOREACH EXECUTE PROCEDURE statement.	Identifier, p. 4-113
<i>procedure</i>	Name of the procedure to execute	The procedure must exist.	Database Object Name, p. 4-25
<i>procedure_var</i>	Name of a procedure variable in the calling or executing procedure	The data type of <i>procedure_var</i> must be appropriate for the value that is being returned.	Identifier, p. 4-113

Usage

A FOREACH loop is the procedural equivalent of using a cursor. When a FOREACH statement executes, the database server takes the following actions:

1. It declares and implicitly opens a cursor.
2. It obtains the first row from the query that is contained within the FOREACH loop, or it obtains the first set of values from the called procedure.
3. It assigns each variable in the variable list the value of the corresponding value from the active set that the SELECT statement or the called procedure creates.
4. It executes the statement block.
5. It fetches the next row from the SELECT statement or called procedure on each iteration, and it repeats step 3.
6. It terminates the loop when it finds no more rows that satisfy the SELECT statement or called procedure. It closes the implicit cursor when the loop terminates.

Because the statement block can contain additional FOREACH statements, cursors can be nested. No limit exists to the number of cursors that can be nested.

A procedure that returns more than one row or set of values is called a *cursory procedure*.

The following 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 further rows.
- The cursor is a select cursor without a HOLD specification, and a transaction completes using COMMIT or ROLLBACK statements.
- An EXIT statement executes, which transfers control out of the FOREACH statement.
- An exception occurs that is not trapped inside the body of the FOREACH statement. (See the ON EXCEPTION statement on [page 3-34](#).)
- A cursor in the calling procedure that is executing this cursory procedure (within a FOREACH loop) closes for any reason.

Using a SELECT...INTO Statement

The SELECT statement in the FOREACH statement must include the INTO clause. It can also include UNION and ORDER BY clauses, but it cannot use the INTO TEMP clause. The syntax of a SELECT statement is shown on [page 2-450](#).

The type and count of each variable in the variable list must match each value that the SELECT...INTO statement returns.

Hold Cursors

The WITH HOLD keyword specifies that the cursor should remain open when a transaction closes (is committed or rolled back).

Updating or Deleting Rows Identified by Cursor Name

Use the WHERE CURRENT OF *cursor* clause to update or delete the current row of *cursor*.

Calling a Procedure in the FOREACH Loop

The called procedure can return zero or more rows.

The type and count of each variable in the variable list must match each value that the called procedure returns.

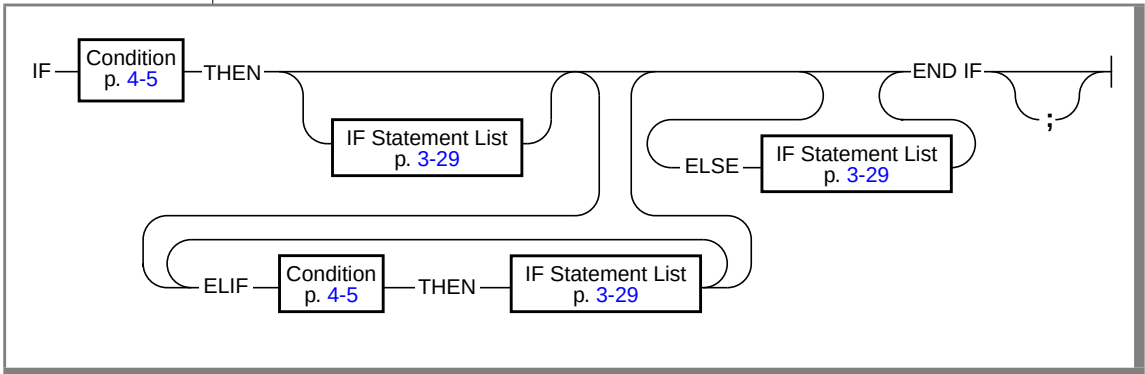
Subset of Expressions Allowed in the Procedure Parameters

You can use any expression as a procedure parameter except an aggregate expression. If you use a subquery or procedure call, the subquery or procedure must return a single value of the appropriate data type and size. For the full syntax of an expression, see [page 4-33](#).

IF

Use an IF statement to create a branch within a procedure.

Syntax



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 ELSE keyword execute.

In the following example, the procedure 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 PROCEDURE 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 PROCEDURE -- str_compare
```

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.

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

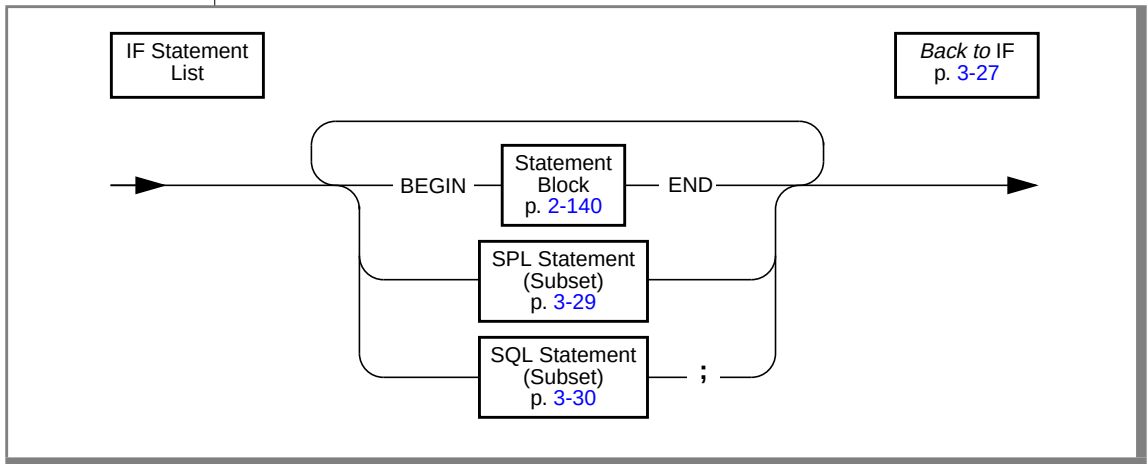
Conditions in an IF statement are evaluated in the same way as conditions in a WHILE statement.

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 SPL Statements Allowed in the Statement Block

You can use any of the following SPL statements in the IF statement list:

CALL	LET
CONTINUE	RAISE EXCEPTION
EXIT	RETURN
FOR	SYSTEM
FOREACH	TRACE
IF	WHILE

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:

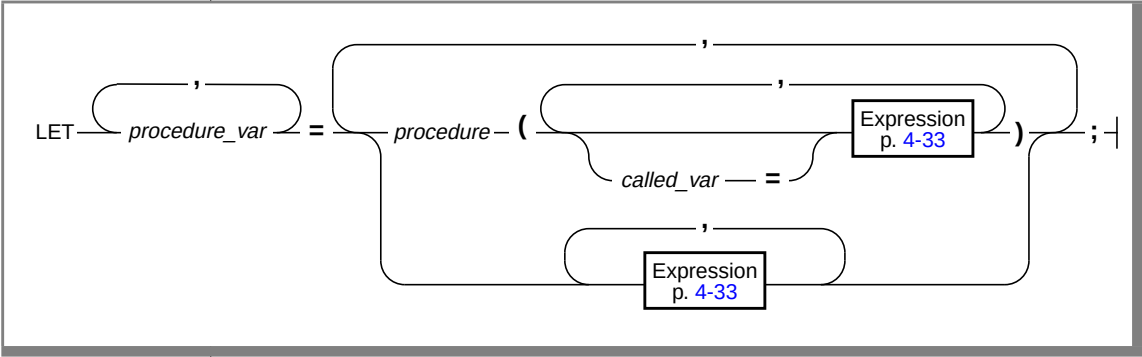
ALLOCATE DESCRIPTOR	FREE
CLOSE DATABASE	GET DESCRIPTOR
CONNECT	GET DIAGNOSTICS
CREATE DATABASE	INFO
CREATE PROCEDURE	LOAD
DATABASE	OPEN
DEALLOCATE DESCRIPTOR	OUTPUT
DECLARE	PREPARE
DESCRIBE	PUT
DISCONNECT	SET CONNECTION
EXECUTE	SET DESCRIPTOR
EXECUTE IMMEDIATE	UNLOAD
FETCH	WHENEVER
FLUSH	

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. You also can use the LET statement to call a procedure within a procedure and assign the returned values to variables.

Syntax



Element	Purpose	Restrictions	Syntax
<i>called_var</i>	Procedure variable of the called procedure	Name or position, but not both, binds procedure arguments to procedure parameters. That is, you can use <i>called_var</i> = syntax for all or none of the arguments that are specified in a LET statement.	Identifier, p. 4-113
<i>procedure</i>	Name of a stored procedure	The stored procedure must exist.	Database Object Name, p. 4-25
<i>procedure_var</i>	Procedure variable	The procedure variable must be defined in the procedure and valid in the statement block.	Identifier, p. 4-113

Usage

If you assign a value to a single variable, it is called a *simple assignment*; if you assign values to two or more variables, it is called a *compound assignment*.

At run time, the value of the SPL expression is computed first. The resulting value is converted to *procedure_var* data type, if possible, and the assignment occurs. If conversion is not possible, an error occurs, and the value of *procedure_var* is undefined.

A compound assignment assigns multiple expressions to multiple variables. The count and data type of expressions in the expression list must match the count and data type of the corresponding variables in the variable list.

The following example shows several LET statements that assign values to procedure 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

Using a SELECT statement in a LET statement is equivalent to using a SELECT...INTO *procedure_var* statement in a procedure. The examples in this section use a SELECT statement in a LET statement. You can use a SELECT statement to assign values to one or more variables on the left side of the = operator, as the following example shows:

```
LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
LET a,b,c = (SELECT c1,c2 FROM t WHERE id = 1), 15;
```

You cannot use a SELECT statement to make multiple values operate on other values. The code in 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.

Calling a Procedure in a LET Statement

You can call a procedure in a LET statement and assign the returned values to variables. If the LET statement includes a procedure call, it invokes the named procedure. You must specify all the necessary arguments to the procedure in the LET statement unless the procedure has default values for its arguments.

If you use the *called_var* = syntax for one of the parameters in the called procedure, you must use it for all the parameters.

The *procedure_var* receives the returned value from a procedure call. A procedure can return more than one value into a list of variable names. You must enclose a procedure that returns more than one row in a FOREACH loop.

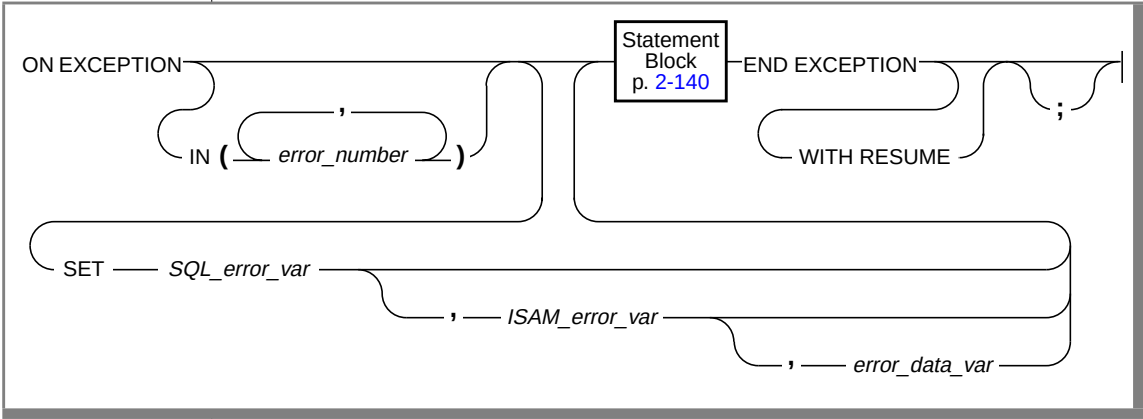
The following example shows several LET statements. The first two are valid LET statements that contain procedure calls. The third LET statement is not legal because it tries to add the output of two procedures and then assign the sum to two variables, *a* and *b*. You can easily split this LET statement into two legal LET statements.

```
LET a, b, c = proc1(name = 'grok', age = 17);
LET a, b, c = 7, proc ('orange', 'green');
LET a, b = proc1() + proc2(); -- ILLEGAL CODE
```

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
<i>error_data_var</i>	Procedure variable that contains a string returned by an SQL error or a user-defined exception	Must be a character data type to receive the error information. Must be valid in the current statement block.	Identifier, p. 4-113
<i>error_number</i>	SQL error number, or an error number created by a RAISE EXCEPTION statement, that is to be trapped	Must be of integer data type. Must be valid in the current statement block.	Literal number, p. 4-139
<i>ISAM_error_var</i>	Variable that receives the ISAM error number of the exception raised	Must be of integer data type. Must be valid in the current statement block.	Identifier, p. 4-113
<i>SQL_error_var</i>	Variable that receives the SQL error number of the exception raised	Must be a character data type. Must be valid in the current statement block.	Identifier, p. 4-113

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 you want to trap as the stored procedure 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 a procedure.

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 procedure. The following procedure inserts a set of values into a table. If the table does not exist, it is created, and the values are inserted. The procedure also returns the total number of rows in the table after the insert occurs.

```

CREATE PROCEDURE 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 PROCEDURE

```

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 procedure, 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. When an error occurs, the database server searches for the last declaration of the ON EXCEPTION statement that traps the particular error code.

```
CREATE PROCEDURE ex_test ()
  DEFINE error_num INT;
  .
  .
  .
  ON EXCEPTION
  SET error_num
  -- action C
  END EXCEPTION

  ON EXCEPTION IN (-300)
  -- action B
  END EXCEPTION
  ON EXCEPTION IN (-210, -211, -212)
  SET error_num
  -- action A
  END EXCEPTION
  .
  .
  .
```

A summary of the sequence of statements in the previous example would be: 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.

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_var*, any error text that the database server returns is put into the *error_data_var*. Error text includes information such as the offending table or column name.

Forcing Continuation of the Procedure

The example on [page 3-36](#) 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 procedure, 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 no statement or block, but only the procedure, contains the ON EXCEPTION statement, the procedure executes a RETURN statement with no arguments to terminate. That is, the procedure 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

```
RAISE EXCEPTION — SQL_error_var — , — ISAM_error_var — , — error_text_var — ;
```

Element	Purpose	Restrictions	Syntax
<i>error_text_var</i>	Procedure variable that contains the error text	The procedure variable must be a character data type and must be valid in the statement block.	Identifier, p. 4-113
<i>ISAM_error_var</i>	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. 4-33
<i>SQL_error_var</i>	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. 4-33

Usage

Use the RAISE EXCEPTION statement to simulate an error or to generate an error with a custom message. An ON EXCEPTION statement can trap the generated error.

If you omit the *ISAM_error_var* parameter, the database server sets the ISAM error code to zero when the exception is raised. If you want to use the *error_text_var* parameter but not specify a value for *ISAM_error_var*, you can specify 0 as the value of *ISAM_error_var*.

The statement can raise either system-generated exceptions or user-generated exceptions.

For example, the following statement raises the error number -208 and inserts the text a missing file into the variable of the system-generated error message:

```
RAISE EXCEPTION -208, 0, 'a missing file';
```

Special Error Numbers

The special error number -746 allows you to produce a customized message. For example, the following statement raises the error number -746 and returns the stated text:

```
RAISE EXCEPTION -746, 0, 'You broke the rules';
```

In the following example, a negative value for alpha raises exception -746 and provides a specific message describing the problem. The code should contain an ON EXCEPTION statement that traps for an exception of -746.

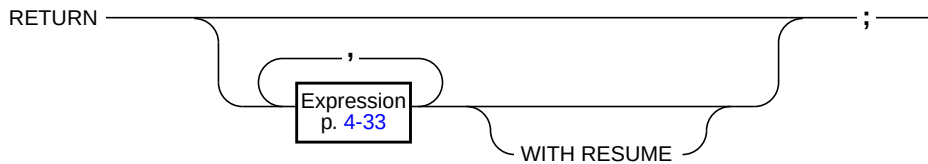
```
FOREACH SELECT c1 INTO alpha FROM sometable
IF alpha < 0 THEN
RAISE EXCEPTION -746, 0, 'a < 0 found' -- emergency exit
END IF
END FOREACH
```

For more information about the scope and compatibility of exceptions, See the ON EXCEPTION statement.

RETURN

Use the RETURN statement to designate the values that the procedure returns to the calling module.

Syntax



Usage

The RETURN statement returns zero or more values to the calling process.

All the RETURN statements in the procedure must be consistent with the RETURNING clause of the CREATE PROCEDURE statement, which the procedure defines. 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 PROCEDURE statement. You can choose to return no values even if you specify one or more values in the RETURNING clause. If you use a RETURN statement without any expressions, but the calling procedure 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 procedure includes two acceptable RETURN statements. A program that calls this procedure should check if no values are returned and act accordingly.

```
CREATE PROCEDURE 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 PROCEDURE
```

A RETURN statement without any expressions exits only if the procedure is declared not to return values; otherwise it returns nulls.

WITH RESUME Keyword

If you use the WITH RESUME keyword after the RETURN statement executes, the next invocation of this procedure (upon the next FETCH OR FOREACH statement) starts from the statement that follows the RETURN statement. If a procedure executes a RETURN WITH RESUME statement, a FOREACH loop in the calling procedure or program must call the procedure.

If a procedure executes a RETURN WITH RESUME statement, a FETCH statement in an ESQL/C application can call it. ♦

E/C

The following example shows a cursory procedure that another procedure can call. After the RETURN I WITH RESUME statement returns each value to the calling procedure, the next line of **sequence** executes the next time **sequence** is called. If **backwards** equals 0, no value is returned to the calling procedure, and execution of **sequence** stops.

```
CREATE PROCEDURE sequence (limit INT, backwards INT)
  RETURNING INT;
  DEFINE i INT;

  FOR i IN (1 TO limit)
    RETURN i WITH RESUME;
  END FOR

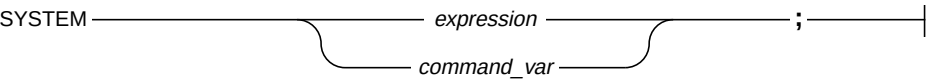
  IF backwards = 0 THEN
    RETURN;
  END IF

  FOR i IN (limit TO 1)
    RETURN i WITH RESUME;
  END IF
END PROCEDURE -- sequence
```

SYSTEM

Use the SYSTEM statement to make an operating-system command run from within a procedure.

Syntax



Element	Purpose	Restrictions	Syntax
<i>command_var</i>	Procedure variable that contains a valid operating-system command	The procedure variable must be a character data type variable that is valid in the statement block.	Identifier, p. 4-113
<i>expression</i>	Any expression that is a user-executable operating-system command	You cannot specify that the command run in the background.	Operating-system dependent

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 procedure statement.

Your procedure 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 DBA- and owner-privileged procedures that contain SYSTEM statements, the operating-system command runs with the permissions of the user who is executing the procedure.

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 the [Informix Guide to SQL: Reference](#).

UNIX

Examples of the SYSTEM Statement on UNIX

The following example shows a SYSTEM statement in a stored procedure. The SYSTEM statement in this procedure causes the UNIX 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; --sensitive_update2
```

WIN NT

Example of the SYSTEM Statement on Windows NT

The following example shows a SYSTEM statement in a stored procedure. The first SYSTEM statement in this procedure causes the Windows NT operating system to send an error message to a temporary file and to put the message in a system log that is sorted alphabetically. The second SYSTEM statement in the procedure causes the operating system to delete the temporary file.

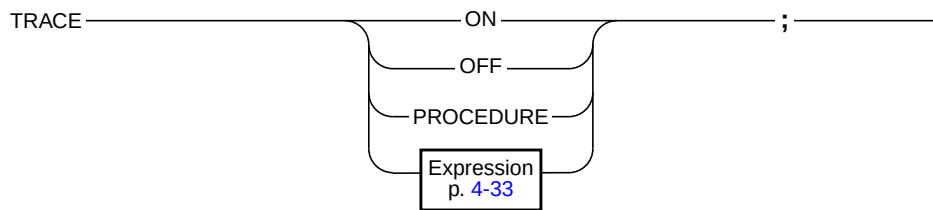
```
CREATE PROCEDURE test_proc()
  .
  .
  .
  SYSTEM 'type errormess101 > %tmp%tmpfile.txt |
        sort >> %SystemRoot%systemlog.txt';
  SYSTEM 'del %tmp%tmpfile.txt';
  .
  .
  .
END PROCEDURE; --test_proc
```

The expressions that follow the SYSTEM statements in this example contain two variables, **%tmp%** and **%SystemRoot%**. Both of these variables are defined by the Windows NT operating system.

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
- Procedure 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.

Called procedures inherit the trace state. That is, a called procedure assumes the same trace state (ON, OFF, or PROCEDURE) as the calling procedure. The called procedure can set its own trace state, but that state is not passed back to the calling procedure.

A procedure 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 procedure calls and statements in the body of the procedure.

TRACE OFF

If you specify the keyword OFF, all tracing is turned off.

TRACE PROCEDURE

If you specify the keyword PROCEDURE, only the procedure calls and return values, but not the body of the procedure, are traced.

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 procedure. 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 using a TRACE OFF statement. The example uses UNIX file-naming conventions.

```
CREATE PROCEDURE tracing ()
  DEFINE i INT;
  BEGIN
    ON EXCEPTION IN (1)
  END EXCEPTION; -- do nothing
  SET DEBUG FILE TO '/tmp/foo.trace';
  TRACE OFF;
  TRACE 'Forloop starts';
  FOR i IN (1 TO 1000)
  BEGIN
```

```

        TRACE 'FOREACH starts';
        FOREACH SELECT...INTO a FROM t
            IF <some condition> THEN
                RAISE EXCEPTION 1      -- emergency exit
            END IF
        END FOREACH

        -- return some value
    END
END FOR

-- do something
END;
END PROCEDURE

```

Example Showing Different Forms of TRACE

The following example shows several different forms of the TRACE statement. The example uses Windows NT file-naming conventions.

```

CREATE PROCEDURE testproc()
    DEFINE i INT;

    SET DEBUG FILE TO 'C:\tmp\test.trace';
    TRACE OFF;
    TRACE 'Entering foo';

    TRACE PROCEDURE;
    LET i = testtoo();

    TRACE ON;
    LET i = i + 1;

    TRACE OFF;
    TRACE 'i+1 = ' || i+1;
    TRACE 'Exiting testproc';

    SET DEBUG FILE TO 'C:\tmp\test2.trace';

END PROCEDURE

```

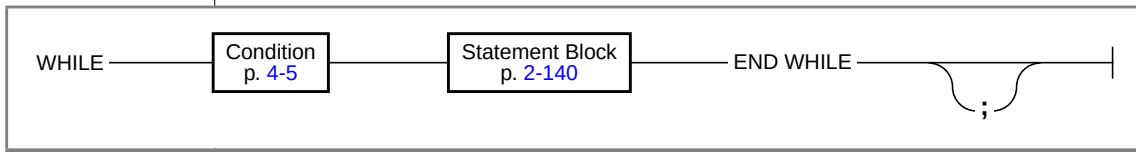
Looking at the Traced Output

To see the traced output, use an editor or utility to display or read the contents of the file.

WHILE

Use the WHILE statement to establish an indefinite loop in a procedure.

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 procedure variables, an immediate error results. In this case, the loop terminates, and an exception is raised.

Example of WHILE Loops in a Stored Procedure

The following example illustrates the use of WHILE loops in a stored procedure. The first WHILE loop executes a DELETE statement. The second WHILE loop executes an INSERT statement and increments the value of a procedure variable.

```
CREATE PROCEDURE simp_while()
  DEFINE i INT;
  WHILE EXISTS (SELECT fname FROM customer
    WHERE customer_num > 400)
    DELETE FROM customer WHERE id_2 = 2;
  END WHILE;

  LET i = 1;
  WHILE i < 10
    INSERT INTO tab_2 VALUES (i);
    LET i = i + 1;
  END WHILE;
END PROCEDURE
```

WHILE

Segments

Condition.	4-5
Database Name.	4-22
Database Object Name	4-25
Data Type.	4-27
DATETIME Field Qualifier.	4-31
Expression	4-33
Identifier	4-113
INTERVAL Field Qualifier	4-130
Literal DATETIME	4-133
Literal INTERVAL.	4-136
Literal Number.	4-139
Optimizer Directives	4-141
Owner Name	4-154
Quoted String	4-157
Relational Operator	4-161

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. Segment descriptions conclude with references to related information in this and other manuals.

Use of Segment Descriptions

The syntax diagram within each segment description is not a stand-alone diagram. Instead it is a subdiagram that is subordinate to the syntax diagram for an SQL or SPL statement.

Syntax diagrams for SQL or SPL statements refer to segment descriptions in two ways:

- A subdiagram-reference box in the syntax diagram for a statement can refer to a segment name and the page number on which the segment description begins.
- The syntax column of the table beneath a syntax diagram can refer to a segment name 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.

For example, if 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. The table beneath the diagram refers to the Database Object Name segment for the syntax for *view*.

The subdiagram for the Database Object 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

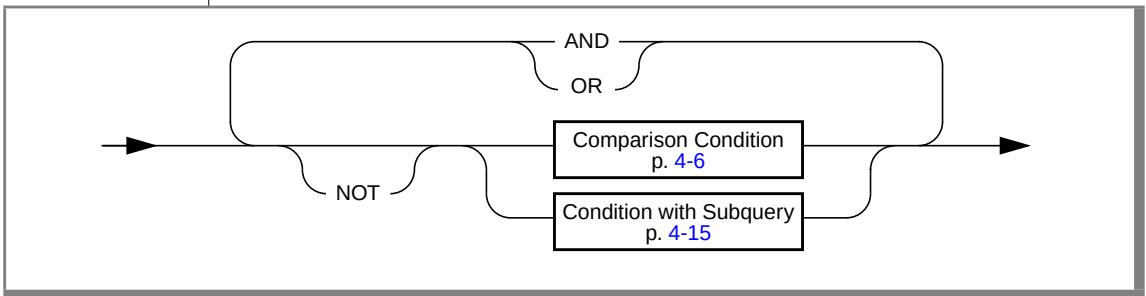
This section describes the following segments.

Condition	Literal INTERVAL
Database Name	Literal Number
Database Object Name	Optimizer Directives
Data Type	Owner Name
DATETIME Field Qualifier	Quoted String
Expression	Relational Operator
Identifier	
INTERVAL Field Qualifier	
Literal DATETIME	

Condition

Use a condition to test 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



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.

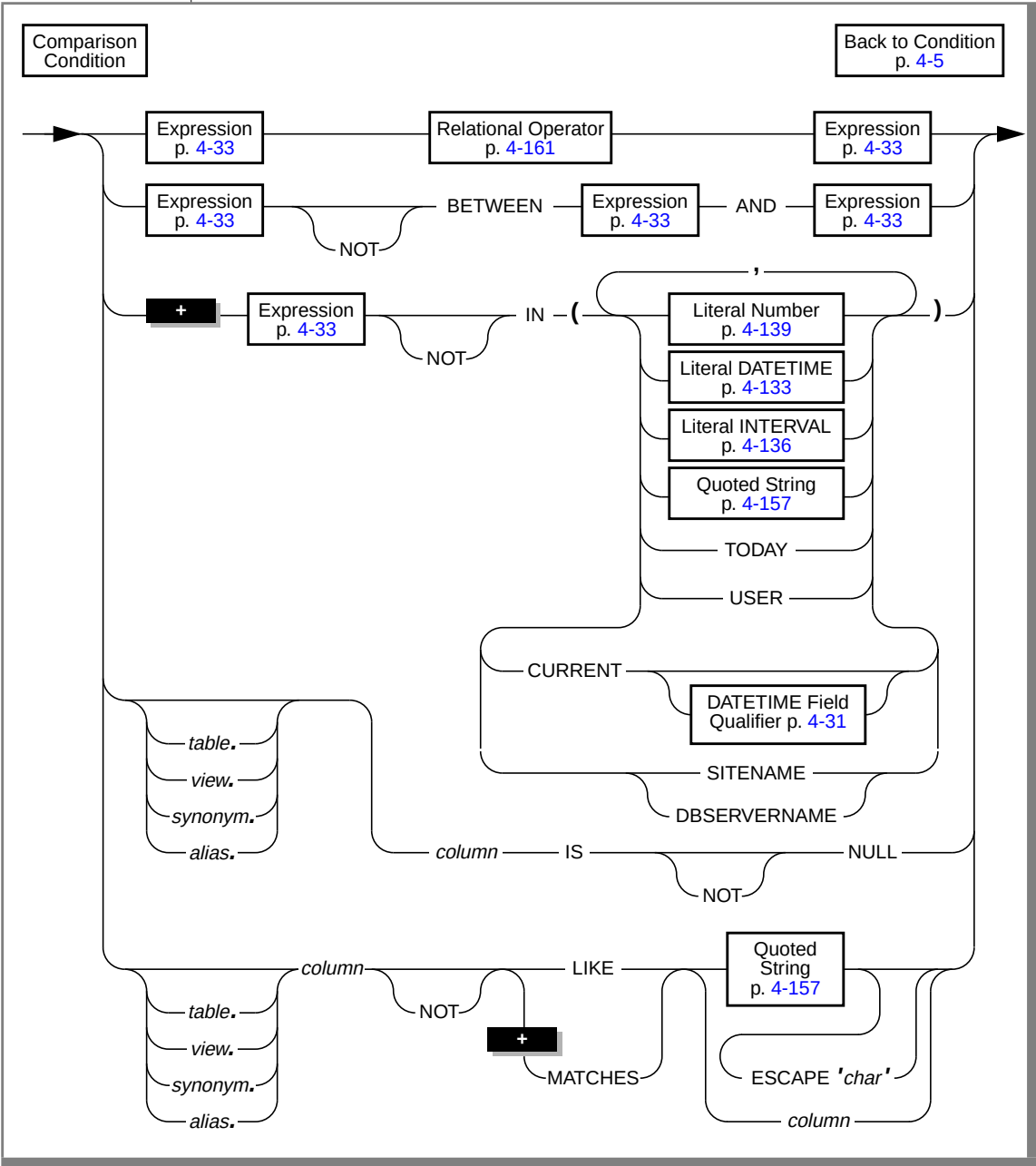
NOT Operator Option

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.



Element	Purpose	Restrictions	Syntax
<i>alias</i>	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. 4-113
<i>char</i>	Single ASCII character to be used as the escape character in the quoted string in a LIKE or MATCHES condition	See “ESCAPE with LIKE” on page 4-14 and “ESCAPE with MATCHES” on page 4-14 .	Quoted String, p. 4-157
<i>column</i>	Name of a column that is used in an IS NULL condition or in a LIKE or MATCHES condition For more information on the meaning of <i>column</i> in these conditions, see “IS NULL Condition” on page 4-11 and “LIKE and MATCHES Condition” on page 4-12 .	The column must exist in the specified table.	Identifier, p. 4-113
<i>synonym</i>	Name of the synonym in which the specified column occurs	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table in which the specified column occurs	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view in which the specified column occurs	The view must exist.	Database Object Name, p. 4-25

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 4-10](#).
- For the BETWEEN condition, refer to [“BETWEEN Condition” on page 4-10](#).
- For the IN condition, refer to [“IN Condition” on page 4-11](#).
- For the IS NULL condition, refer to [“IS NULL Condition” on page 4-11](#).
- For the LIKE and MATCHES condition, refer to [“LIKE and MATCHES Condition” on page 4-12](#).



For a discussion of the different types of comparison conditions in the context of the `SELECT` statement, see [“Using a Condition in the WHERE Clause” on page 2-472](#).

Warning: When you specify a date value in a comparison 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 comparison condition. When you specify a 2-digit year, the **DBCENTURY** environment variable can affect how the database server interprets the comparison condition, so the comparison condition might not work as you intended. For more information on the **DBCENTURY** environment variable, see the [“Informix Guide to SQL: Reference.”](#)

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 > '05/01/98'
AND ship_weight < 30
```


Relational-Operator Condition

Some relational-operator conditions are shown in the following examples:

```
city[1,3] = 'San'

o.order_date > '6/12/98'

WEEKDAY(paid_date) = WEEKDAY(CURRENT-31 UNITS day)

YEAR(ship_date) < YEAR (TODAY)

quantity <= 3

customer_num <> 105

customer_num != 105
```

If either expression is null for a row, the condition evaluates to false. For example, if **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 4-11](#).

```
SELECT customer_num, order_date FROM orders
    WHERE paid_date IS NULL
```

BETWEEN Condition

For a BETWEEN test to be true, the value of the expression on the left of the BETWEEN keyword must be in the inclusive range of the values of the two expressions on the right of the BETWEEN keyword. Null values do not satisfy the condition. You cannot use NULL for either expression that defines the range.

Some BETWEEN conditions are shown in the following examples:

```
order_date BETWEEN '6/1/97' and '9/7/97'

zipcode NOT BETWEEN '94100' and '94199'

EXTEND(call_dtime, DAY TO DAY) BETWEEN
  (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT

lead_time BETWEEN INTERVAL (1) DAY TO DAY
  AND INTERVAL (4) DAY TO DAY

unit_price BETWEEN loprice AND hiprice
```

IN Condition

The IN condition is satisfied when the expression to the left of the word IN is included in the list of items. 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)
```

E/C

In ESQ/C, 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.

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 clause. You can use the quote character as the escape character in matching any other pattern if you write it as '' ''.

NOT Option

The NOT option 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 Option

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

In an ANSI-compliant database, you can only use an escape character to escape a percent sign (%), an underscore (_), or the escape character itself. ♦

The following condition tests for the string tennis, alone or in a longer string, such as tennis ball or table tennis paddle:

```
WHERE description LIKE '%tennis%'
```

The following condition tests for all descriptions that contain an underscore. The backslash (\) is necessary because the underscore (_) is a wildcard character.

```
WHERE description LIKE '%\_%'
```

MATCHES Option

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 square brackets ([...]) match any of the enclosed characters, including character ranges as in [a - z]. Characters inside the square brackets cannot be escaped.
^	A caret (^) as the first character within the square 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 following condition is true for any name that ends with the letters a, b, c, or d:

```
WHERE fname MATCHES '*[a-d]'
```

ESCAPE with LIKE

The ESCAPE clause 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;
```

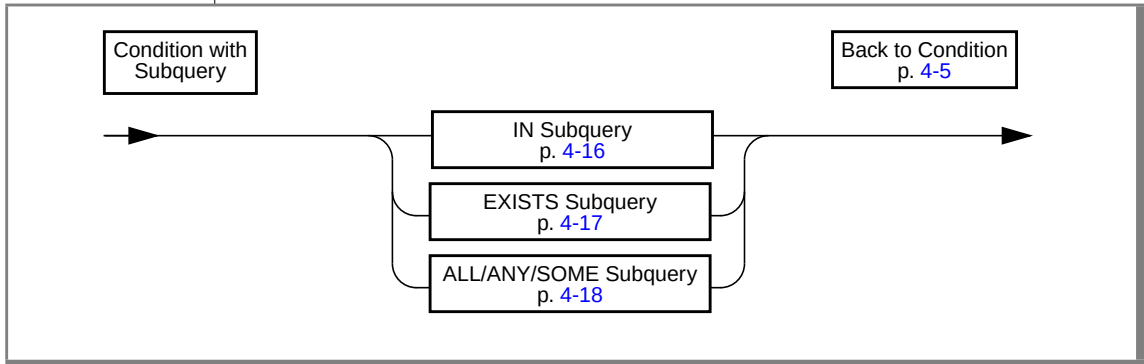
ESCAPE with MATCHES

The ESCAPE clause lets you include a question mark (?), an asterisk (*), and a left or right square 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'
```

Condition with 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:

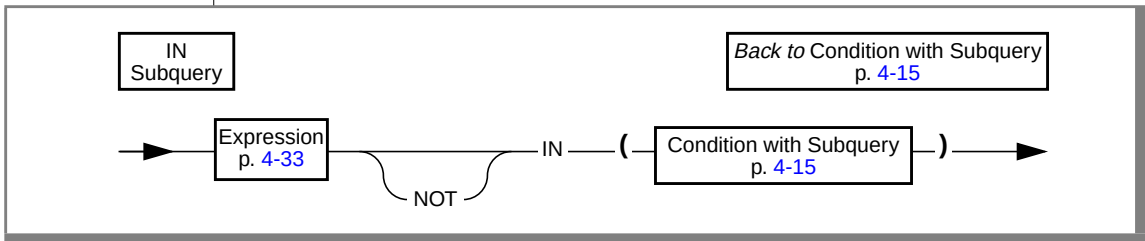
- Compare an expression to the result of another SELECT statement
- Determine whether an expression is included in the results of another SELECT statement
- Ask whether another SELECT statement selects any rows

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. For a discussion of the different kinds of subquery conditions in the context of the SELECT statement, see [“Using a Condition in the WHERE Clause” on page 2-472](#).

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 2-450](#).

IN Subquery

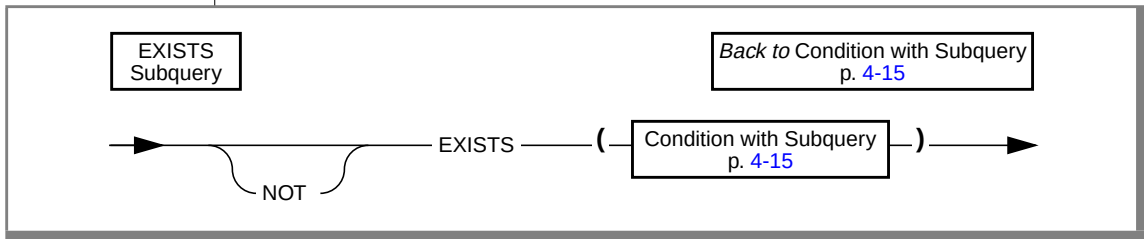


An IN subquery condition is true if the value of the expression matches one or more of the values that the subquery selects. The subquery must return only one column, but it can return more than one row. The keyword IN is equivalent to the =ANY sequence. The keywords NOT IN are equivalent to the !=ALL sequence. See “[ALL / ANY / SOME Subquery](#)” on [page 4-18](#).

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.

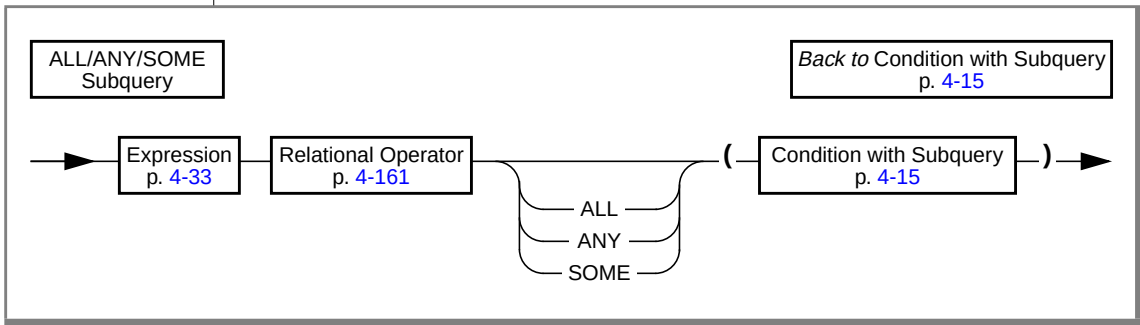
EXISTS Subquery

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

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.

Using the ALL Keyword

The ALL keyword 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.

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)
```

Using the NOT keyword with an ALL subquery tests whether an expression is not true for all subquery values. For example, the following condition is true when the expression **total_price** is not greater than all the selected values. That is, it is true when **total_price** is not greater than the highest total price in order number 1023.

```
NOT total_price > ALL (SELECT total_price FROM items
                       WHERE order_num = 1023)
```

Using the ANY or SOME Keywords

The ANY keyword denotes that the search condition is true if the comparison is true for at least one of the values that is returned. If the subquery returns no value, the search condition is false. The SOME keyword is an alias for ANY.

The following conditions are true when the total price is greater than the total price of at least one of the items in order number 1023. The first condition uses the ANY keyword; the second uses the MIN aggregate function.

```
total_price > ANY (SELECT total_price FROM items
                   WHERE order_num = 1023)
```

```
total_price > (SELECT MIN(total_price) FROM items
               WHERE order_num = 1023)
```

Using the NOT keyword with an ANY subquery tests whether an expression is not true for any subquery value. For example, the following condition is true when the expression **total_price** is not greater than any selected value. That is, it is true when **total_price** is greater than none of the total prices in order number 1023.

```
NOT total_price > ANY (SELECT total_price FROM items
                      WHERE order_num = 1023)
```

Omitting the ANY, ALL, or SOME Keywords

You can omit the keywords ANY, ALL, or SOME in a subquery if you know that the subquery will return exactly one value. If you omit the ANY, ALL, or SOME keywords, and the subquery returns more than one value, you receive an error. The subquery in the following example returns only one row because it uses an aggregate function:

```
SELECT order_num FROM items
       WHERE stock_num = 9 AND quantity =
             (SELECT MAX(quantity) FROM items WHERE stock_num = 9)
```

Conditions with AND or OR

You can combine simple conditions with the logical operators AND or OR to form complex conditions. The following SELECT statements contain examples of complex conditions in their WHERE clauses:

```
SELECT customer_num, order_date FROM orders
  WHERE paid_date > '1/1/97' OR paid_date IS NULL

SELECT order_num, total_price FROM items
  WHERE total_price > 200.00 AND manu_code LIKE 'H%'

SELECT lname, customer_num FROM customer
  WHERE zipcode BETWEEN '93500' AND '95700'
  OR state NOT IN ('CA', 'WA', 'OR')
```

The following truth tables show the effect of the AND and OR operators. The letter T represents a true condition, F represents a false condition, and the question mark (?) represents an unknown value. Unknown values occur when part of an expression that uses a logical operator is null.

AND	T	F	?	OR	T	F	?
T	T	F	?	T	T	T	T
F	F	F	F	F	T	F	?
?	?	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

For discussions of comparison conditions in the SELECT statement and of conditions with a subquery, see the [Informix Guide to SQL: Tutorial](#).

For information on the GLS aspects of conditions, see the [Informix Guide to GLS Functionality](#).

Database Name

Use the Database Name segment to specify 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
<i>dbname</i>	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. The <i>dbname</i> can have a maximum of 18 characters.	Identifier, p. 4-113
<i>dbservername</i>	Name of the database server on which the database that is named in <i>dbname</i> resides.	The database server that is specified in <i>dbservername</i> must exist. You cannot put a space between the @ symbol and <i>dbservername</i> .	Identifier, p. 4-113
<i>db_env_var</i>	Host variable that contains a value representing a database environment	Variable must be a fixed-length character data type.	Name must conform to language-specific rules for variable names.

Usage

Database names are not case sensitive. You cannot use delimited identifiers for a database name. If you are creating a database, the name that you assign to the database can be 18 characters, inclusive.

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 discussion on naming databases in the [Informix Guide to GLS Functionality](#). ♦

Specifying the Database Server

You can choose a database on another database server as your current database by specifying a database server name. The database server that *dbservername* specifies must match the name of a database server that is listed in your **sqlhosts** information.

Using the @ Symbol

The @ symbol is a literal character that introduces the database server name. If you specify a database server name, do not put any spaces between the @ symbol and the database server name. You can either put a space between dbname and the @ symbol, or omit the space.

The following examples show valid database specifications:

```
empinfo@personnel  
empinfo @personnel
```

In these examples, **empinfo** is the name of the database and **personnel** is the name of the database server.

Using a Path-Type Naming Method

If you use a path-type naming method, do not put spaces between the quotes, slashes, and names, as the following example shows:

```
'//personnel/empinfo'
```

In this example, **empinfo** is the name of the database and **personnel** is the name of the database server.

The maximum length of the database name and directory path, including *dbservername*, is 128 characters.

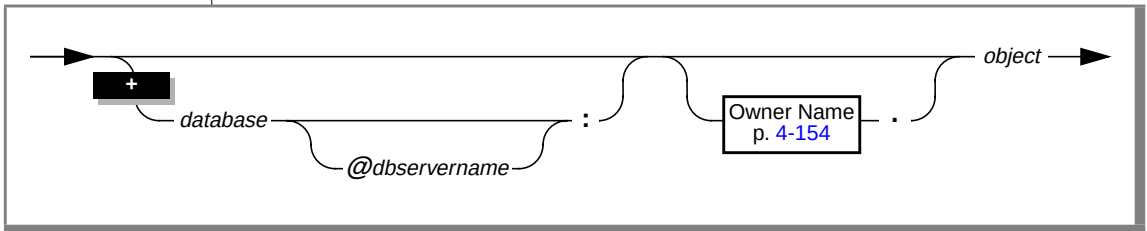
Using a Host Variable

You can use a host variable within an ESQL/C application to contain a value that represents a database environment.

Database Object Name

Use the database object name segment to specify the name of a constraint, index, procedure, trigger, table, synonym, or view. Use this segment whenever you see a reference to database object name.

Syntax



Element	Purpose	Restrictions	Syntax
<i>database</i>	Name of the database where the database object resides	The database must exist.	Identifier, p. 4-113
<i>dbservername</i>	Name of the database server where the database resides	The database server must exist. You cannot put a space between the @ symbol and <i>dbservername</i> .	Identifier, p. 4-113
<i>object</i>	Name of a database object in the database	If you are creating the database object, character limitations exist. For more information, see “Identifier” on page 4-113 . If you are accessing the database object, the database object must exist.	Identifier, p. 4-113

Usage

If you are creating or renaming a database object, the name that you specify must be unique in relation to other database objects of the same type in the database. For example, a new constraint name must be unique among constraint names that exist in the database.

ANSI

A new name of a table, synonym, or view, must be unique among all the tables, synonyms, views, and temporary tables that already exist in the database.

In an ANSI-compliant database, the *ownername.object* combination must be unique in a database.

A database object name must include the owner name for a database object that you do not own. For example, if you specify a table that you do not own, you must specify the owner of the table also. The owner of all the system catalog tables is **informix**. ♦

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in database object names. For more information, see the [Informix Guide to GLS Functionality](#). ♦

Procedures and SQL Functions with the Same Names

If you create a procedure with the same name as an SQL function and then explicitly define that name as a procedure, any calls by that name are to the procedure instead of the SQL function. That is, you cannot use the system function in the statement block in which the procedure is defined.

Fully qualified identifier

A database object name that includes the owner name, database name, and database server name is called a fully qualified identifier.

The following example shows a fully qualified table name:

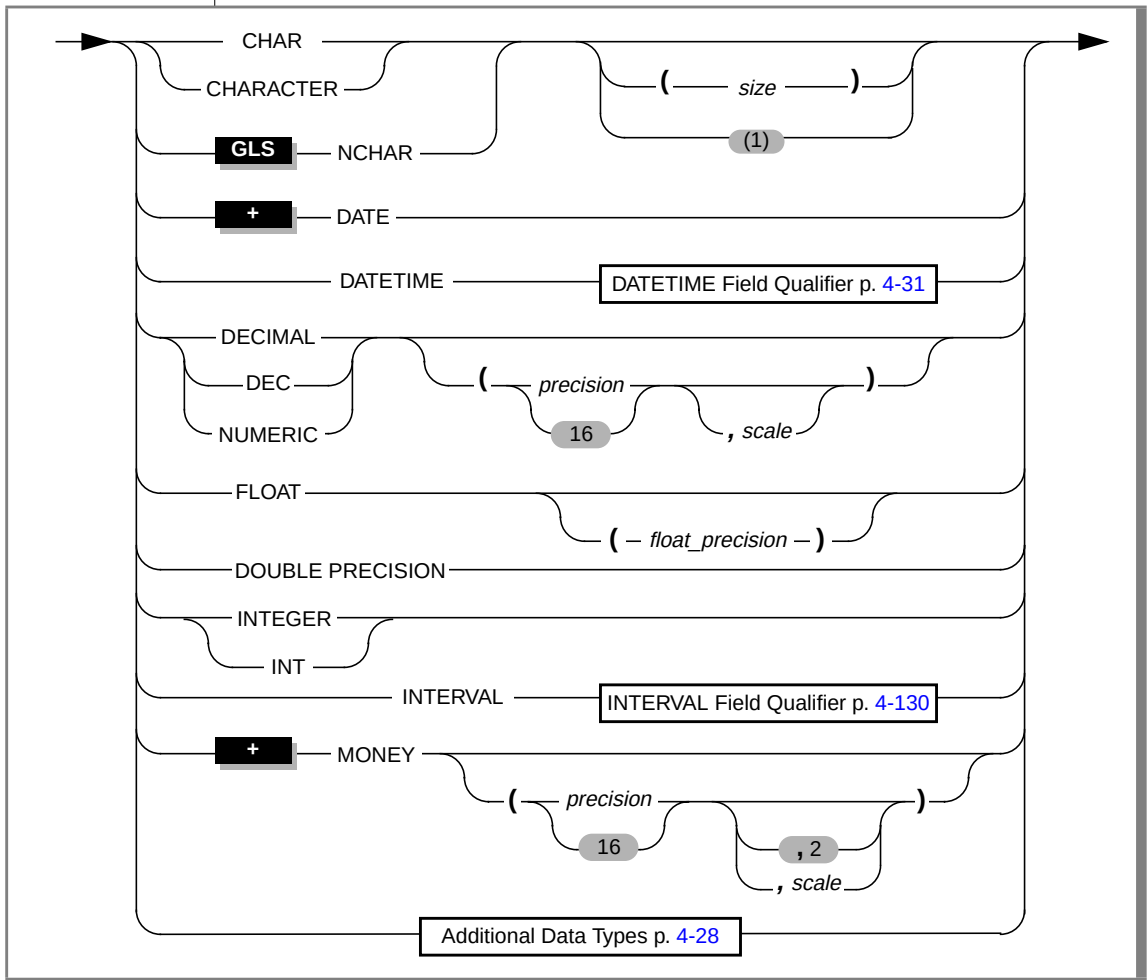
```
empinfo@personnel:markg.emp_names
```

In this example, the name of the database is **empinfo**. The name of the database server is **personnel**. The name of the owner of the table is **markg**. The name of the table is **emp_names**.

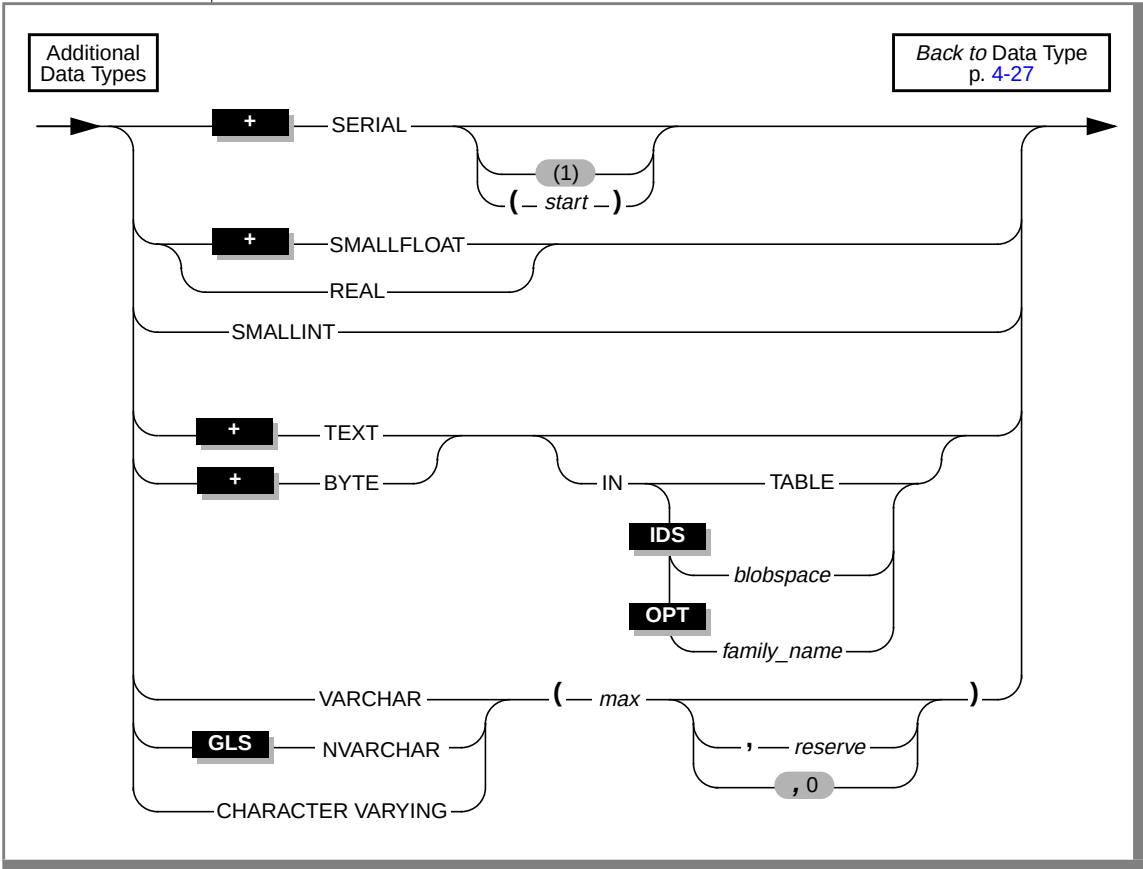
Data Type

Use the Data Type segment to specify 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



Additional Data Types



Element	Purpose	Restrictions	Syntax
<i>blob space</i>	Name of an existing blob space	The blob space must exist.	Identifier, p. 4-113
<i>family_name</i>	Quoted string constant that specifies a family name or variable name in the optical family	The family name or variable name must exist. For additional information about optical families, see the Guide to the Optical Subsystem .	Quoted String, p. 4-157
<i>float_precision</i>	Float precision is ignored	You must specify a positive integer.	Literal Number, p. 4-139

Element	Purpose	Restrictions	Syntax
<i>max</i>	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 <i>max</i> is 254 bytes.	Literal Number, p. 4-139
<i>precision</i>	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. 4-139
<i>reserve</i>	Amount of space in bytes reserved for a CHARACTER VARYING, NVARCHAR or VARCHAR column even if the actual number of bytes stored in the column is less than <i>reserve</i> The default value of <i>reserve</i> is 0.	You must specify an integer value between 0 and 255 bytes. However, the value you specify for <i>reserve</i> must be less than the value you specify for <i>max</i> .	Literal Number, p. 4-139
<i>scale</i>	Number of digits to the right of the decimal point	You must specify an integer between 1 and <i>precision</i> .	Literal Number, p. 4-139
<i>size</i>	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. 4-139
<i>start</i>	Starting number for values in a SERIAL column	You must specify an integer greater than 0 and less than 2,147,483,647.	Literal Number, p. 4-139

(2 of 2)

For more information, see the discussion of all data types in the [Informix Guide to SQL: Reference](#).

Fixed and Varying Length Data Types

The data type CHAR is for fixed-length character data. Use the ANSI-compliant CHARACTER VARYING data type to specify varying length character data. You can also specify varying length data with the Informix VARCHAR data type.

NCHAR and NVARCHAR Data Types

For a discussion of the NCHAR and NVARCHAR data types, see the [Informix Guide to GLS Functionality](#). ♦

References

For discussions of individual data types, see the [Informix Guide to SQL: Reference](#).

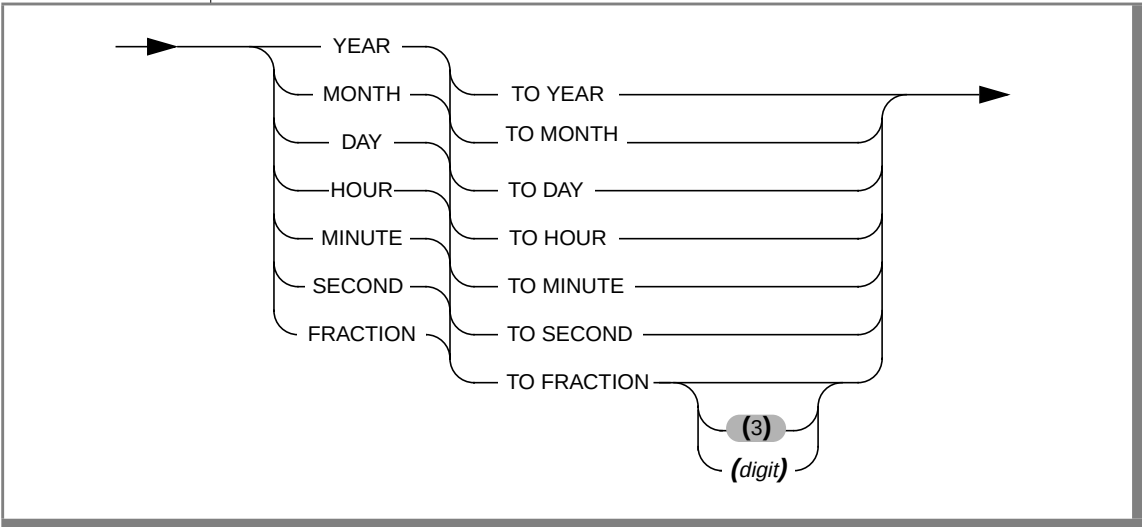
For a general discussion of data types, see the [Informix Guide to Database Design and Implementation](#).

For a discussion of the NCHAR and NVARCHAR data types and the GLS aspects of other character data types, see the [Informix Guide to GLS Functionality](#).

DATETIME Field Qualifier

Use a DATETIME field qualifier to specify 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
<i>digit</i>	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. 4-139

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 column.

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
HOURL	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
```

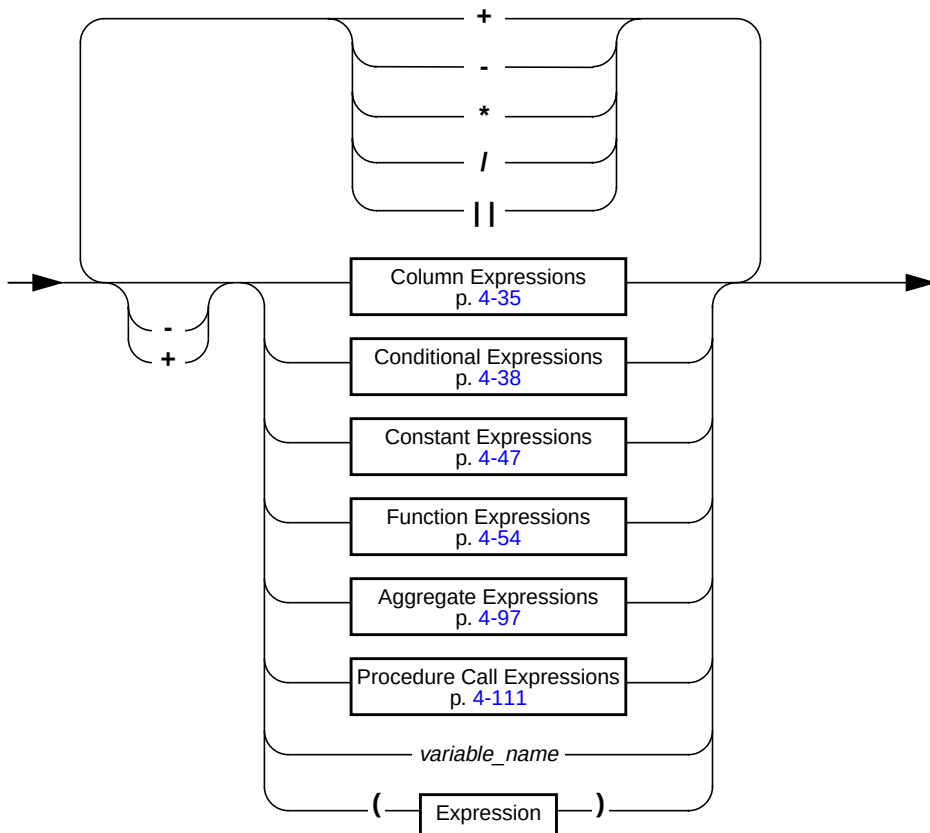
References

For an explanation of the DATETIME field qualifier, see the discussion of the DATETIME data type in the [Informix Guide to SQL: Reference](#).

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



Element	Purpose	Restrictions	Syntax
<i>variable_name</i>	Host variable or procedure variable The value stored in the variable is one of the expression types shown in the syntax diagram.	The expression that is stored in <i>variable_name</i> must conform to the rules for expressions of that type.	Name must conform to language-specific rules for variable names. Identifier, p. 4-113

Usage

To combine expressions, connect them with arithmetic operators for addition, subtraction, multiplication, and division.

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.

Concatenation Operator

You can use the concatenation operator (||) to concatenate two expressions. For example, 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
```

E/C

You cannot use the concatenation operator in an embedded-language-only statement. The ESQL/C-only statements appear in the following list:

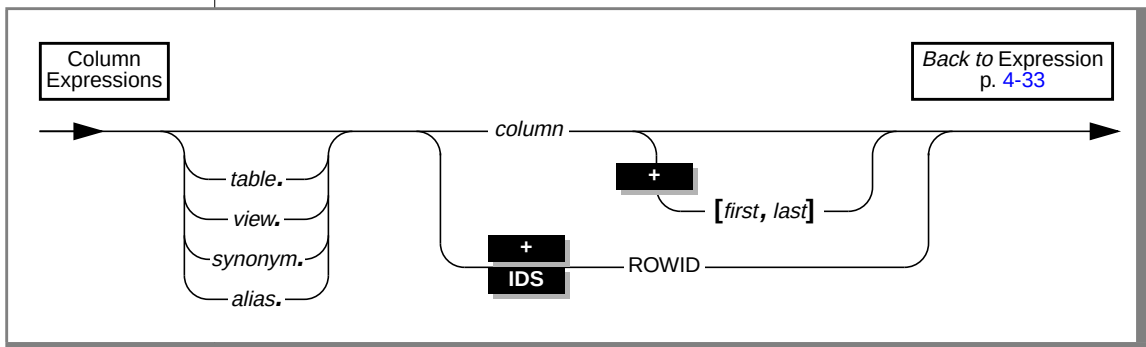
ALLOCATE DESCRIPTOR	FREE
CLOSE	GET DESCRIPTOR
CREATE PROCEDURE FROM	GET DIAGNOSTICS
DEALLOCATE DESCRIPTOR	OPEN
DECLARE	PREPARE
DESCRIBE	PUT
EXECUTE	SET CONNECTION
EXECUTE IMMEDIATE	SET DESCRIPTOR
FETCH	WHENEVER
FLUSH	

You can use the concatenation operator in the SELECT, INSERT, or EXECUTE PROCEDURE statement in the DECLARE statement.

You can use the concatenation operator in the SQL statement or statements in the PREPARE statement. ♦

Column Expressions

The possible syntax for column expressions is shown in the following diagram.



Element	Purpose	Restrictions	Syntax
<i>alias</i>	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 <i>alias</i> occurs.	Identifier, p. 4-113
<i>column</i>	Name of the column that you are specifying	The restrictions depend on the statement in which <i>column</i> occurs.	Identifier, p. 4-113
<i>first</i>	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. 4-139
<i>last</i>	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. 4-139
<i>synonym</i>	Name of the synonym in which the specified column occurs	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table in which the specified column occurs	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view in which the specified column occurs	The view must exist.	Database Object Name, p. 4-25

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 Subscripts on Character Columns

You can use subscripts on **CHAR**, **VARCHAR**, **NCHAR**, **NVARCHAR**, **BYTE**, and **TEXT** columns. The subscripts indicate the starting and ending character positions 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]
```

For information on the GLS aspects of column subscripts and substrings, see the [Informix Guide to GLS Functionality](#). ♦

GLS

IDS

Using Rowids

In Dynamic Server, you can use the rowid column that is associated with a table row as a property of the row. The rowid column is essentially a hidden column in nonfragmented tables and in fragmented tables that were created with the `WITH ROWIDS` clause. The rowid column is unique for each row, but it is not necessarily sequential. 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

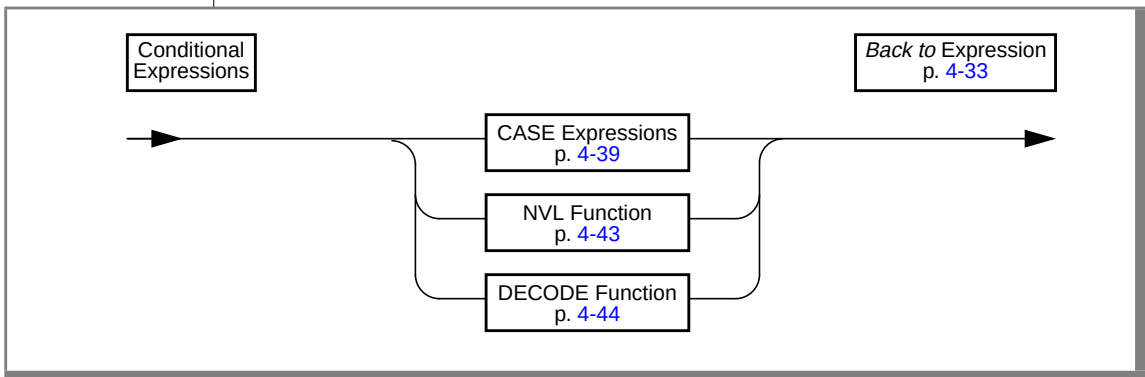
SELECT HEX(rowid) FROM customer
WHERE customer_num = 106
```

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 the ROWID keyword in the select list of a query that contains an aggregate function.

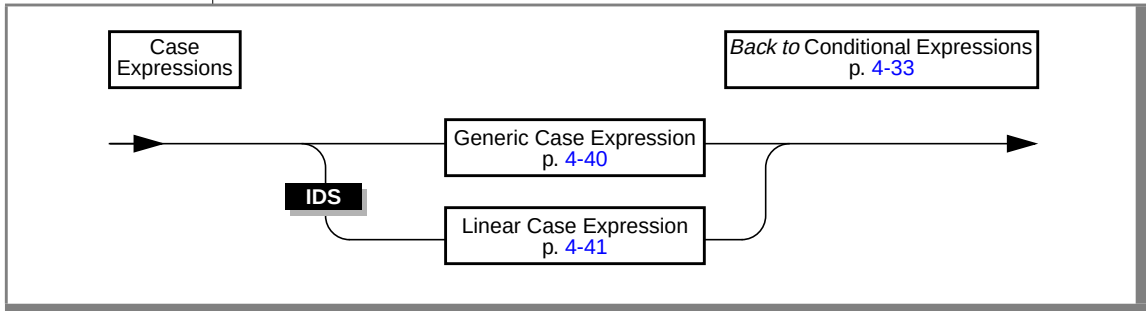
Conditional Expressions

Conditional expressions return values that depend on the outcome of conditional tests. The following diagram shows the syntax for Conditional Expressions.



CASE Expressions

The CASE expression allows an SQL statement such as the SELECT statement to return one of several possible results, depending on which of several condition tests evaluates to true. The CASE expression has two forms as the following diagram shows: generic CASE expressions and linear CASE expressions.



Using CASE Expressions

You can use a generic or linear CASE expression wherever you can use a column expression in an SQL statement (for example, in the select list of a SELECT statement.) You must include at least one WHEN clause in the CASE expression. Subsequent WHEN clauses and the ELSE clause are optional.

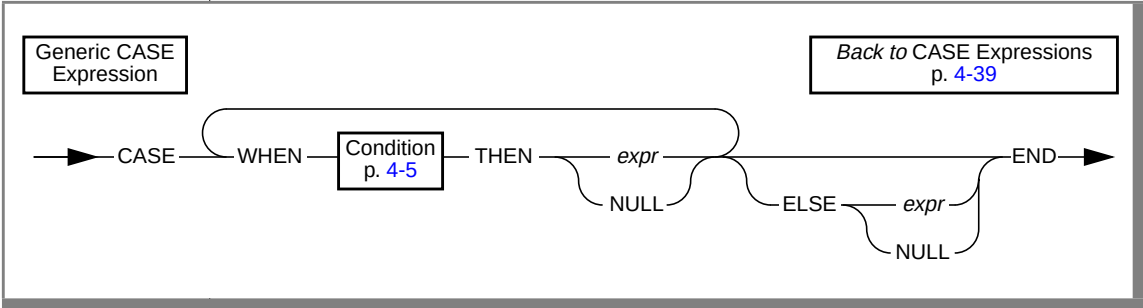
The expressions in the search condition or the result value expression can contain subqueries.

You can nest a CASE expression in another CASE expression.

When a CASE expression appears in an aggregate expression, you cannot use aggregate functions in the CASE expression.

Generic CASE Expressions

A generic CASE expression tests for a true condition in a WHEN clause and when it finds a true condition it returns the result specified in the THEN clause.



Element	Purpose	Restrictions	Syntax
<i>expr</i>	Expression that returns a result value of a certain data type	The data type of <i>expr</i> in a THEN clause must be compatible with the data types of other value expressions in other THEN clauses.	Expression, p. 4-33

The database server processes the WHEN clauses in the order that they appear in the statement. As soon as the database server finds a WHEN clause whose search condition evaluates to true, it takes the corresponding result value expression as the overall result of the CASE expression, and it stops processing the CASE expression.

If no WHEN condition evaluates to true, the database server takes the result of the ELSE clause as the overall result. If no WHEN condition evaluates to true, and no ELSE clause was specified, the resulting value is null. You can use the IS NULL condition to handle null results. For information on how to handle null values, see “IS NULL Condition” on page 4-11.

The following example shows the use of a generic CASE expression in the select list of a SELECT statement. In this example the user retrieves the name and address of each customer as well as a calculated number that is based on the number of problems that exist for that customer.

```
SELECT cust_name,
       CASE
         WHEN number_of_problems = 0
           THEN 100
         WHEN number_of_problems > 0 AND number_of_problems < 4
           THEN number_of_problems * 500
         WHEN number_of_problems >= 4 and number_of_problems <= 9
           THEN number_of_problems * 400
         ELSE
           (number_of_problems * 300) + 250
       END,
       cust_address
FROM custtab
```

In a generic CASE expression, all the results should be of the same type, or they should evaluate to a common compatible type. If the results in all the WHEN clauses are not of the same type, or if they do not evaluate to values of mutually compatible types, an error occurs.

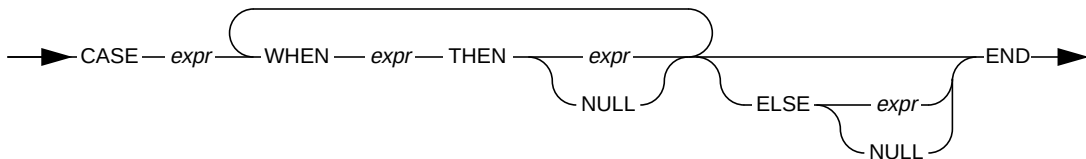
IDS

Linear CASE Expressions

In Dynamic Server, a linear CASE expression tests for a match between the value expression that follows the CASE keyword and a value expression in a WHEN clause.

Linear CASE
Expression

[Back to CASE Expressions
p. 4-39](#)



Element	Purpose	Restrictions	Syntax
<i>expr</i>	Expression that evaluates to a value of a certain data type or that returns a result value of a certain data type	The data type of the <i>expr</i> that follows the WHEN keyword in a WHEN clause must be compatible with the data type of the value expression that follows the CASE keyword. The data type of <i>expr</i> in a THEN clause must be compatible with the data types of other value expressions in other THEN clauses.	Expression, p. 4-33

First the database server evaluates the value expression that follows the CASE keyword. Then the database server processes the WHEN clauses in the order that they appear in the CASE expression. As soon as the database server finds a WHEN clause where the value expression after the WHEN keyword evaluates to the same value as the value expression that follows the CASE keyword, it takes the value expression that follows the THEN keyword as the overall result of the CASE expression. Then the database server stops processing the CASE expression.

If none of the value expressions that follow the WHEN keywords evaluates to the same value as the value expression that follows the CASE keyword, the database server takes the result value expression of the ELSE clause as the overall result of the CASE expression. If all of the value expressions that follow the WHEN keyword in all the WHEN clauses do not evaluate to the same value as the value expression that follows the CASE keyword, and the user did not specify an ELSE clause, the resulting value is null.

The following example shows a linear CASE expression in the select list of a SELECT statement. For each movie in a table of movie titles, the SELECT statement displays the title of the movie, the cost of the movie, and the type of movie. The statement uses a CASE expression to derive the type of each movie.

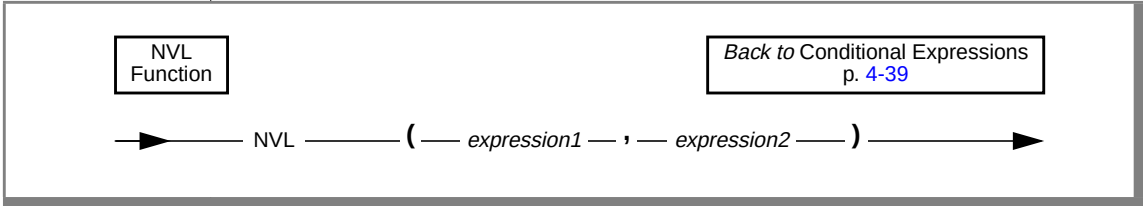
```
SELECT title,
       CASE movie_type
         WHEN 1 THEN 'HORROR'
         WHEN 2 THEN 'COMEDY'
         WHEN 3 THEN 'ROMANCE'
         WHEN 4 THEN 'WESTERN'
         ELSE 'UNCLASSIFIED'
       END,
       our_cost
FROM movie_titles
```

In linear CASE expressions, the types of value expressions in all the WHEN clauses have to be compatible with the type of the value expression that follows the CASE keyword.

IDS

NVL Function

In Dynamic Server, the NVL expression returns different results depending on whether its first argument evaluates to null.



Element	Purpose	Restrictions	Syntax
expression1 expression2	Any expression that evaluates to a value of a certain data type or that returns a result value of a certain data type	The expression cannot be a host variable or a BYTE or TEXT data type. The expression1 and expression2 values must evaluate to a compatible data type.	Expression, p. 4-33

NVL evaluates *expression1*. If *expression1* is not null, NVL returns the value of *expression1*. If *expression1* is null, NVL returns the value of *expression2*. The expressions *expression1* and *expression2* can be of any data type, as long as they evaluate to a common compatible type.

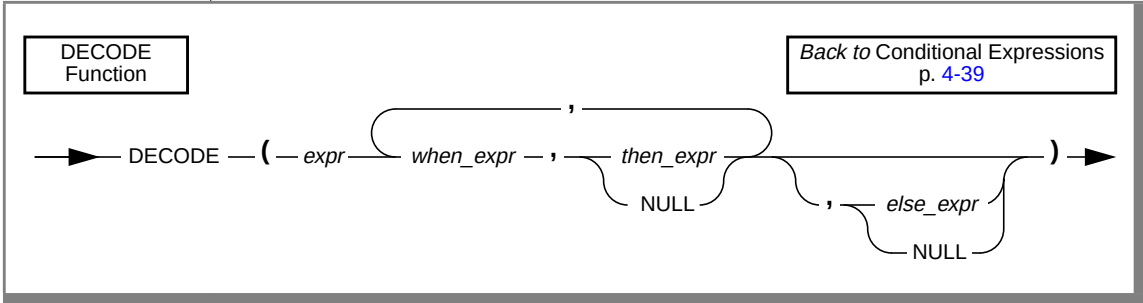
Suppose that the **addr** column of the **employees** table has null values in some rows, and the user wants to be able to print the label Address unknown for these rows. The user enters the following SELECT statement to display the label Address unknown when the **addr** column has a null value.

```
SELECT fname, NVL (addr, 'Address unknown') AS address
FROM employees
```

IDS

DECODE Function

In Dynamic Server, the DECODE expression is similar to the CASE expression in that it can print different results depending on the values found in a specified column.



Element	Purpose	Restrictions	Syntax
<i>expr</i> <i>else_expr</i> <i>then_expr</i> <i>when_expr</i>	Expression that evaluates to a value of a certain data type or that returns a result value of a certain data type	The data type of <i>when_expr</i> must be compatible with the data type of <i>expr</i> .	Expression, p. 4-33

The expressions *expr*, *when_expr*, and *then_expr* are required. DECODE evaluates *expr* and compares it to *when_expr*. If the value of *when_expr* matches the value of *expr*, DECODE returns *then_expr*.

The expressions *when_expr* and *then_expr* are an expression pair, and you can specify any number of expression pairs in the DECODE function. In all cases, DECODE compares the first member of the pair against *expr* and returns the second member of the pair if the first member matches *expr*.

If no expression matches *expr*, DECODE returns *else_expr*. However, if no expression matches *expr* and the user did not specify *else_expr*, DECODE returns NULL.

You can specify any data type as input, but two limitations exist.

- The parameters *expr*, *when_expr*, *then_expr*, and *else_expr* all must have the same data type, or they must evaluate to a common compatible type.
- All occurrences of the parameters *then_expr* must have the same data type, or they must evaluate to a common compatible type. Similarly, all occurrences of *then_expr* must have the same data type as *else_expr*, or they must evaluate to a common compatible type.

Suppose that a user wants to convert descriptive values in the **evaluation** column of the **students** table to numeric values in the output. The following table shows the contents of the **students** table.

firstname	evaluation
Edward	Great
Joe	not done
Mary	Good
Jim	Poor

The user now enters a SELECT statement with the DECODE function to convert the descriptive values in the **evaluation** column to numeric equivalents.

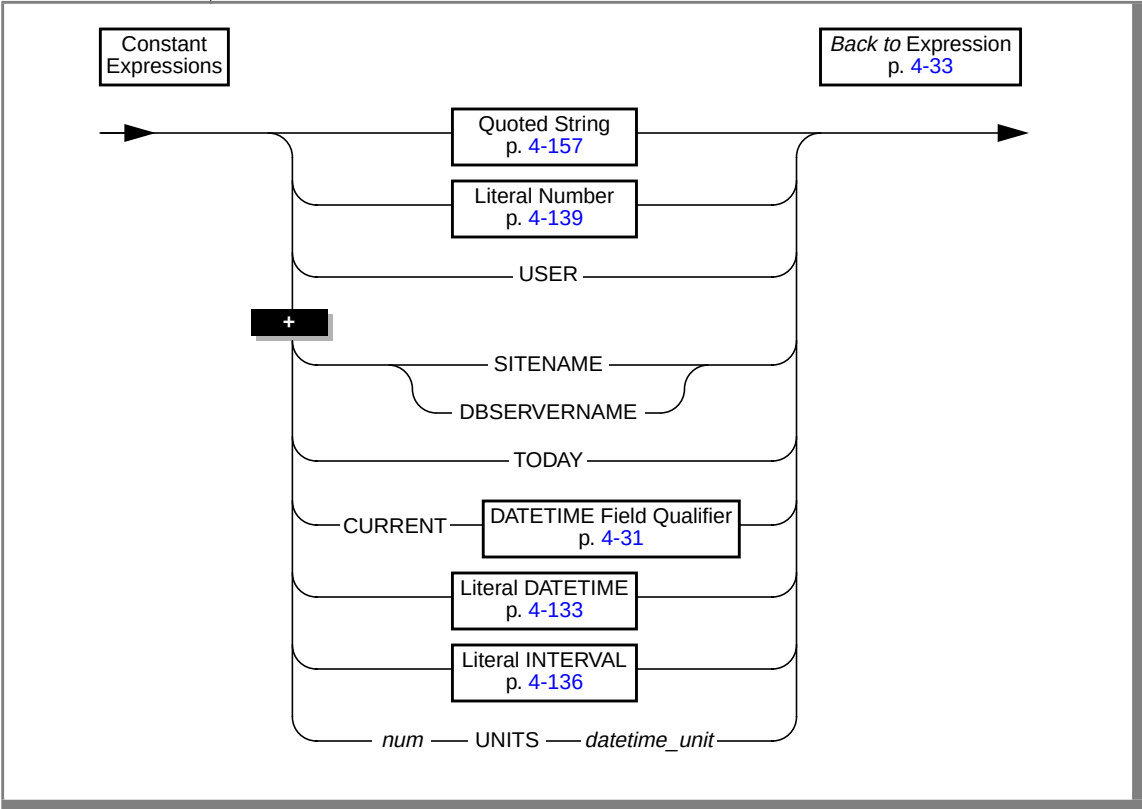
```
SELECT firstname, DECODE(evaluation,
    'Poor', 0,
    'Fair', 25,
    'Good', 50,
    'Very Good', 75,
    'Great', 100,
    -1) as grade
FROM students
```

The following table shows the output of this SELECT statement.

firstname	grade
Edward	100
Joe	-1
Mary	50
Jim	0

Constant Expressions

The following diagram shows the possible syntax for constant expressions.



Element	Purpose	Restrictions	Syntax
<i>datetime_unit</i>	Unit that specifies 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.
<i>num</i>	Literal number that you use to specify the number of datetime units For more information on this parameter, see “UNITS Keyword” on page 4-53 .	If <i>n</i> is not an integer, it is rounded down to the nearest whole number when it is used. The value that you specify for <i>n</i> must be appropriate for the datetime unit that you choose.	Literal Number, p. 4-139

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
```

The following list provides references for further information:

- For quoted strings as expressions, see [“Quoted String as an Expression” on page 4-49](#).
- For the USER function in an expression, see [“USER Function” on page 4-49](#).
- For the SITENAME and DBSERVERNAME functions in an expression, refer to [“DBSERVERNAME and SITENAME Functions” on page 4-50](#).
- For literal numbers as expressions, see [“Literal Number as an Expression” on page 4-51](#).

- For the TODAY function in an expression, see [“TODAY Function” on page 4-51](#).
- For the CURRENT function in an expression, see [“CURRENT Function” on page 4-51](#).
- For literal DATETIME as an expression, see [“Literal DATETIME as an Expression” on page 4-52](#).
- For literal INTERVAL as an expression, see [“Literal INTERVAL as an Expression” on page 4-53](#).
- For the UNITS keyword in an expression, see [“UNITS Keyword” on page 4-53](#).

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 = '1997-1-1 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. ♦

DBSERVERNAME and SITENAME Functions

The DBSERVERNAME function returns the database server name, as defined in the ONCONFIG file for the installation where the current database resides or as specified in the **INFORMIXSERVER** environment variable. The two function names, DBSERVERNAME and SITENAME 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, 'HR0', 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 4-133](#)). 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 1997 to the current instant.

```
DELETE FROM cust_calls WHERE
    res_dtime < CURRENT YEAR TO MINUTE

SELECT * FROM orders WHERE DAY(ord_date) < DAY(CURRENT)

SELECT * FROM cust_calls WHERE call_dtime
    BETWEEN '1997-1-1 00:00:00' AND CURRENT
```

Literal DATETIME as an Expression

The following examples show literal DATETIME as an expression:

```
SELECT DATETIME (1997-12-6) YEAR TO DAY FROM customer

UPDATE cust_calls SET res_dtime = DATETIME (1998-07-07 10:40)
    YEAR TO MINUTE
    WHERE customer_num = 110
    AND call_dtime = DATETIME (1998-07-07 10:24) YEAR TO MINUTE

SELECT * FROM cust_calls
    WHERE call_dtime
        = DATETIME (1998-12-25 00:00:00) YEAR TO SECOND
```

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.

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.

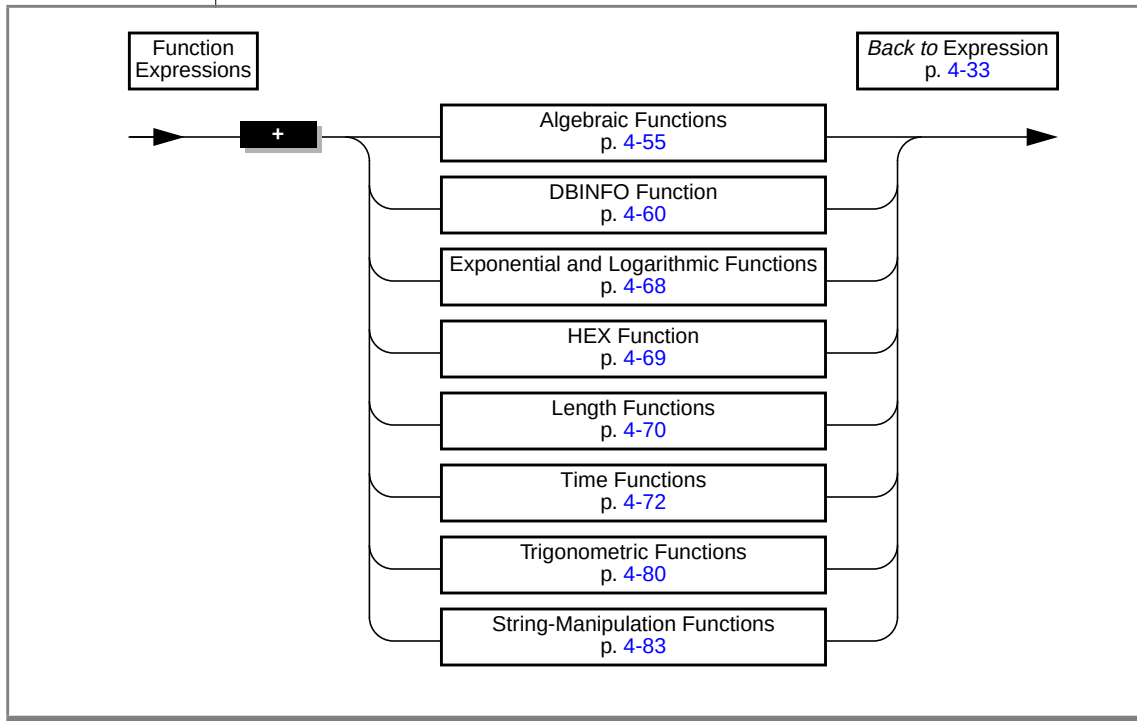
```
SELECT lead_time + 5 UNITS DAY FROM manufact
```

```
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'
```

Function Expressions

A function expression takes an argument, as the following diagram shows.



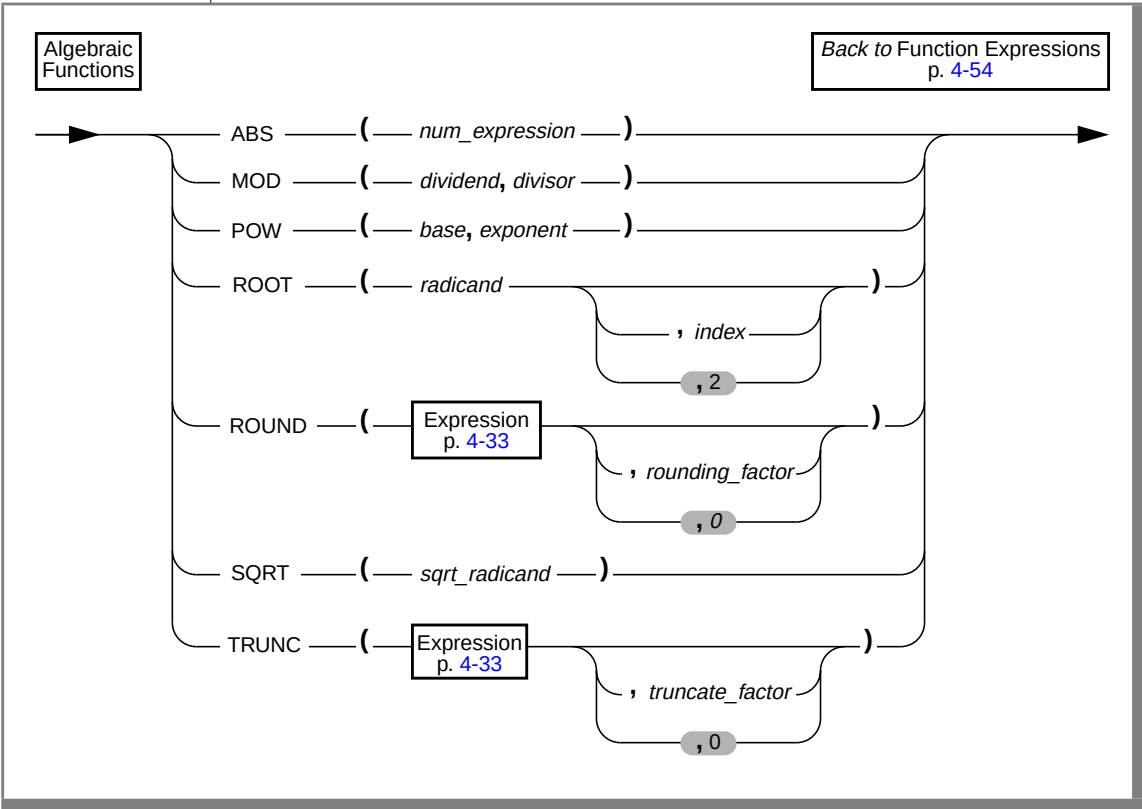
The following examples show function expressions:

```
EXTEND (call_dtime, YEAR TO SECOND)
MDY (12, 7, 1900 + cur_yr)
DATE (365/2)
LENGTH ('abc') + LENGTH (pvar)
HEX (customer_num)
HEX (LENGTH(123))
TAN (radians)
```

ABS (- 32)
EXP (3)
MOD (10 , 3)

Algebraic Functions

An algebraic function takes one or more arguments, as the following diagram shows.



Element	Purpose	Restrictions	Syntax
<i>base</i>	Value to be raised to the power that is specified in <i>exponent</i> The <i>base</i> value is the first argument that is supplied to the POW function.	You can enter in <i>base</i> any real number or any expression that evaluates to a real number.	Expression, p. 4-33
<i>dividend</i>	Value to be divided by the value in <i>divisor</i> The <i>dividend</i> value is the first argument supplied to the MOD function.	You can enter in <i>dividend</i> any real number or any expression that evaluates to a real number.	Expression, p. 4-33
<i>divisor</i>	Value by which the value in <i>dividend</i> is to be divided The <i>divisor</i> value is the second argument that is supplied to the MOD function.	You can enter in <i>divisor</i> any real number except zero or any expression that evaluates to a real number other than zero.	Expression, p. 4-33
<i>exponent</i>	Power to which the value that is specified in <i>base</i> is to be raised The <i>exponent</i> value is the second argument that is supplied to the POW function.	You can enter in <i>exponent</i> any real number or any expression that evaluates to a real number.	Expression, p. 4-33
<i>index</i>	Type of root to be returned, where 2 represents square root, 3 represents cube root, and so on The <i>index</i> value is the second argument that is supplied to the ROOT function. The default value of <i>index</i> is 2.	You can enter in <i>index</i> any real number except zero or any expression that evaluates to a real number other than zero.	Expression, p. 4-33
<i>num_expression</i>	Numeric expression for which an absolute value is to be returned The expression serves as the argument for the ABS function.	The value of <i>num_expression</i> can be any real number.	Expression, p. 4-33
<i>radicand</i>	Expression whose root value is to be returned The <i>radicand</i> value is the first argument that is supplied to the ROOT function.	You can enter in <i>radicand</i> any real number or any expression that evaluates to a real number.	Expression, p. 4-33

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>rounding_factor</i>	Number of digits to which a numeric expression is to be rounded The <i>rounding_factor</i> value is the second argument that is supplied to the ROUND function. The default value of <i>rounding_factor</i> is zero. This default means that the numeric expression is rounded to zero digits or the ones place.	The value you specify in <i>rounding_factor</i> must be an integer between +32 and -32, inclusive.	Literal Number, p. 4-139
<i>sqrt_radicand</i>	Expression whose square root value is to be returned The <i>sqrt_radicand</i> value is the argument that is supplied to the SQRT function.	You can enter in <i>sqrt_radicand</i> any real number or any expression that evaluates to a real number.	Expression, p. 4-33
<i>truncate_factor</i>	Position to which a numeric expression is to be truncated The <i>truncate_factor</i> value is the second argument that is supplied to the TRUNC function. The default value of <i>truncate_factor</i> is zero. This default means that the numeric expression is truncated to zero digits or the ones place.	The value you specify in <i>truncate_factor</i> must be an integer between +32 and -32, inclusive. For more information on this restriction, see “TRUNC Function” on page 4-60.	Literal Number, p. 4-139

(2 of 2)

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. The value returned is INT. The following example uses a 30-day billing cycle to determine how far into the billing cycle today is:

```
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 (πr^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 4-1 shows.

Figure 4-1
ROUND Function

Expression:

`ROUND (24,536.8746, -2) = 24,500.00`

`ROUND (24,536.8746, 0) = 24,537.00`

`ROUND (24,536.8746, 2) = 24,536.87`

2 4 5 3 6 . 8 7 4 6

↓ ↓ ↓
-2 0 2

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 4-2 shows.

Figure 4-2
TRUNC Function

Expression:	2 4 5 3 6 . 8 7 4 6
TRUNC (24536.8746, -2) =24500	↓ ↓ ↓
TRUNC (24536.8746, 0) = 24536	-2 0 2
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
```

DBINFO Function

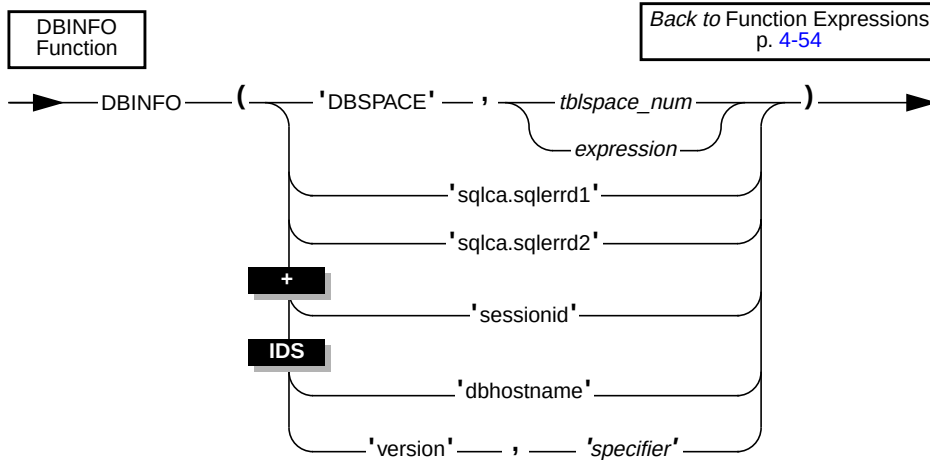
Use the DBINFO function for any of the following purposes:

- 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

IDS

- To find out the session ID of the current session
- In Dynamic Server, to find out the hostname of the database server to which a client application is connected
- In Dynamic Server, to find out the exact version of the database server to which a client application is connected ♦

You can use the DBINFO function anywhere within SQL statements and within stored procedures.



Element	Purpose	Restrictions	Syntax
<i>expression</i>	Expression that evaluates to <i>tblspace_num</i>	The expression can contain procedure variables, host variables, column names, or subqueries, but it must evaluate to a numeric value.	Expression, p. 4-33
<i>specifier</i>	A literal value that specifies which part of the version string is to be returned	For the set of values that you can provide for <i>specifier</i> , see “Using the ‘version’ Option” on page 4-66 .	Expression, p. 4-33
<i>tblspace_num</i>	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. 4-139

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. The following example uses the 'sqlca.sqlerrd1' option:

```
.
EXEC SQL create table fst_tab (ordernum serial, part_num 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, and EXECUTE PROCEDURE statements processed. To ensure valid results, use this option after SELECT and EXECUTE PROCEDURE statements have completed executing. In addition, to ensure valid results when you use this option within cursors, make sure that all rows are fetched before the cursors are closed.

The following example shows 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 fst_tab WHERE part_num=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 the database server, the 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 your [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;
```

IDS

Using the 'dbhostname' Option

In Dynamic Server, you can use the 'dbhostname' option to retrieve the hostname of the database server to which a database client is connected. This option retrieves the physical machine name of the computer on which the database server is running.

In the following example, the user enters the 'dbhostname' option of `DBINFO` in a `SELECT` statement to retrieve the hostname of the database server to which DB-Access is connected:

```
SELECT DBINFO('dbhostname')
FROM systables
WHERE tabid = 1
```


The following table shows the result of this query.

(constant)
rd_lab1

Using the 'version' Option

In Dynamic Server, you can use the 'version' option of the DBINFO function to retrieve the exact version number of the database server against which the client application is running. This option retrieves the exact version string from the message log. The value of the full version string is the same as that displayed by the **-V** option of the **oninit** utility.

You use the *specifier* parameter of the 'version' option to specify which part of the version string you want to retrieve. The following table lists the values you can enter in the *specifier* parameter, shows which part of the version string is returned for each *specifier* value, and gives an example of what is returned by each value of *specifier*. Each example returns part of the complete version string Informix Dynamic Server Version 7.30.UC1.

Value of specifier Parameter	Part of Version String Returned	Example of Return Value
'server-type'	The type of server	Informix Dynamic Server
'major'	The major version number of the current server version	7
'minor'	The minor version number of the current server version	30

Value of specifier Parameter	Part of Version String Returned	Example of Return Value
'os'	The operating-system identifier within the version string: T = Windows NT U = UNIX 32 bit running on a 32-bit operating system H = UNIX 32 bit running on a 64-bit operating system F = UNIX 64 bit running on a 64-bit operating system	U
'level'	The interim release level of the current server version	C1
'full'	The complete version string as it would be returned by oninit -V	Informix Dynamic Server Version 7.30.UC1

The following example shows how to use the 'version' option of DBINFO in a SELECT statement to retrieve the major version number of the database server that the DB-Access client is connected to.

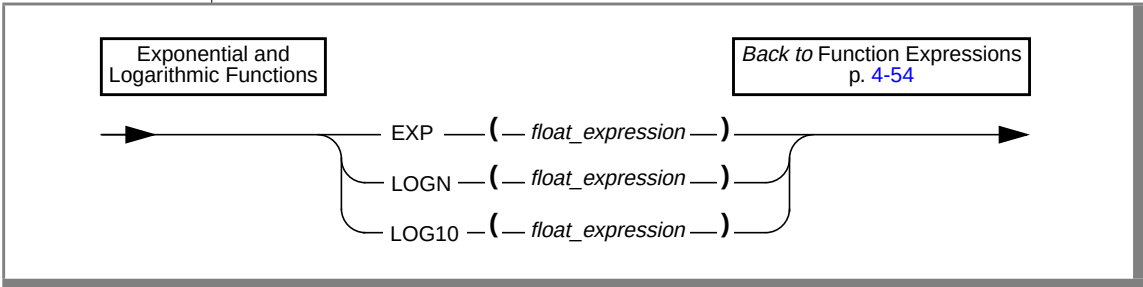
```
SELECT DBINFO('version', 'major')
FROM systables
WHERE tabid = 1
```

The following table shows the result of this query.

(constant)
7

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
<i>float_expression</i>	Expression that serves as an argument to the EXP, LOGN, or LOG10 functions For information on the meaning of <i>float_expression</i> 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. 4-33

EXP Function

The EXP function returns the exponent of a numeric expression. The following example returns the exponent of 3 for each row of the **angles** table:

```
SELECT EXP(3) FROM angles
```

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
int_expression	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. 4-33

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. For more information about encoded information, see the *Informix Guide to SQL: Reference*.

```
SELECT colname, coltype, HEX(collength)
FROM syscolumns C, systables T
WHERE C.tabid = T.tabid AND T.tabname = 'orders'
```

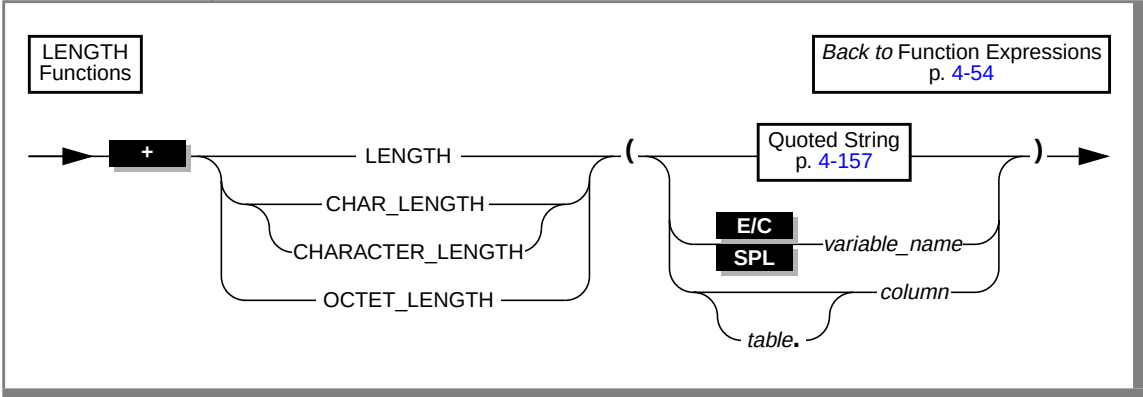
The following example lists the names of all the tables in the current database and their corresponding tblspace number in hexadecimal format. This example is particularly useful because the two most significant bytes in the hexadecimal number constitute the dbspace number. They are used to identify 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 of a column in the specified table	The column must have a character data type.	Identifier, p. 4-113
table	Name of the table in which the specified column occurs	The table must exist.	Database Object Name, p. 4-25
variable_name	Host variable or procedure variable that contains a character string	The host variable or procedure variable must have a character data type.	Name must conform to language-specific rules for variable names. Procedure Call Expressions, p. 4-111

You can use length functions to determine the length of a column, string, or variable. The length functions are the following:

- ★ LENGTH
- ★ OCTET_LENGTH
- CHAR_LENGTH (also known as CHARACTER_LENGTH)

Each of these functions has a distinct purpose.

The LENGTH Function

The LENGTH function returns the number of bytes in a character column, not including any trailing spaces. With BYTE or TEXT 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
```

E/C

In ESQL/C, 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 [Informix Guide to GLS Functionality](#). ♦

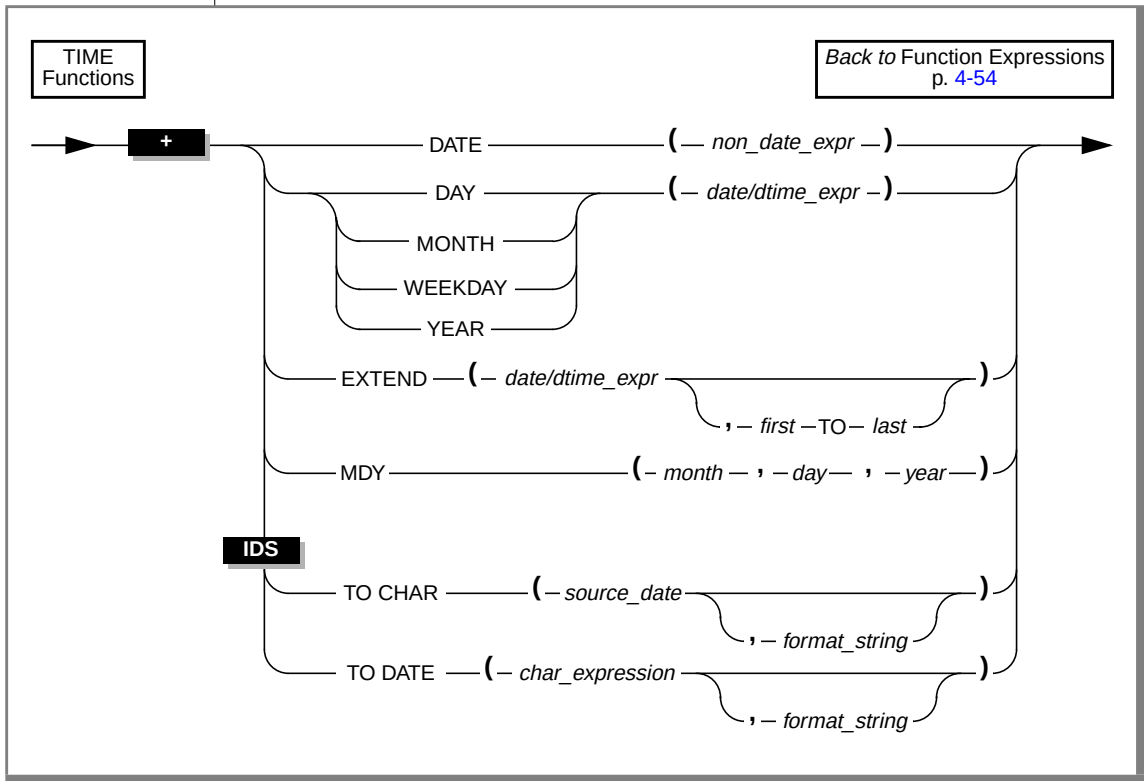
The OCTET_LENGTH Function

The OCTET_LENGTH function returns the number of bytes in a character column, including any trailing spaces. For a discussion of the OCTET_LENGTH function, see the [Informix Guide to GLS Functionality](#).

The CHAR_LENGTH Function

The CHAR_LENGTH function returns the number of characters (not bytes) in a character column. The CHARACTER_LENGTH function is a synonym for the CHAR_LENGTH function. For a discussion of the CHAR_LENGTH function, see the [Informix Guide to GLS Functionality](#).

Time Functions



Element	Purpose	Restrictions	Syntax
<i>char_expression</i>	Expression to be converted to a DATE or DATETIME value	The expression must be of a character data type. It can be a constant, host variable, expression, or column.	Expression, p. 4-33
<i>date/dtime_expr</i>	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. 4-33
<i>day</i>	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. 4-33
<i>first</i>	Qualifier that specifies the first field in the result If you do not specify <i>first</i> and <i>last</i> qualifiers, the default value of <i>first</i> is YEAR.	The qualifier can be any DATETIME qualifier, as long as it is larger than <i>last</i> .	DATETIME Field Qualifier, p. 4-31
<i>format_string</i>	String that represents the format of the DATE or DATETIME value	This string must have a character data type. The string must contain a valid date format, according to the formats allowed in the GL_DATE and GL_DATETIME environment variables. The string can be a column, host variable, expression, or constant.	Quoted String, p. 4-157
<i>last</i>	Qualifier that specifies the last field in the result If you do not specify <i>first</i> and <i>last</i> qualifiers, the default value of <i>last</i> is FRACTION(3).	The qualifier can be any DATETIME qualifier, as long as it is smaller than <i>first</i> .	DATETIME Field Qualifier, p. 4-31
<i>month</i>	Expression that represents the number of the month	The expression must evaluate to an integer between 1 and 12, inclusive.	Expression, p. 4-33

(1 of 2)

Element	Purpose	Restrictions	Syntax
<i>non_date_expr</i>	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. 4-33
<i>source_date</i>	Expression that represents a date that is to be converted to a character string	This value must be of type DATETIME or DATE. It can be a host variable, expression, column, or constant.	Expression, p. 4-33
<i>year</i>	Expression that represents the year	The expression must evaluate to a four-digit integer. You cannot use a two-digit abbreviation.	Expression, p. 4-33

(2 of 2)

DATE Function

The DATE function returns a DATE 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('12/31/97')
```

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('02/01/1998') 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 (1998), the DATE function encounters four digits when it expects a two-digit day (valid day values must be between 01 and 31). It therefore cannot convert the value. For the SELECT statement to execute successfully with the Y2MD/ value for **DBDATE**, the non-date expression would need to be '98/02/01'. For information on the format of **DBDATE**, see 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'
```

TO_CHAR Function

In Dynamic Server, the TO_CHAR function converts a DATE or DATETIME value to a character string. The character string contains the date that was specified in the *source_date* parameter, and represents this date in the format that was specified in the *format_string* parameter.

If the value of the *source_date* parameter is null, the result of the function is a null value.

If you omit the *format_string* parameter, the TO_CHAR function uses the default date format to format the character string. The default date format is specified by environment variables such as **GL_DATETIME** and **GL_DATE**.

The *format_string* parameter does not have to imply the same qualifiers as the *source_date* parameter. When the implied formatting mask qualifier in *format_string* is different from the qualifier in *source_date*, the TO_CHAR function extends the DATETIME value as if it had called the EXTEND function.

In the following example, the user wants to convert the **begin_date** column of the **tab1** table to a character string. The **begin_date** column is defined as a DATETIME YEAR TO SECOND data type. The user uses a SELECT statement with the TO_CHAR function to perform this conversion.

```
SELECT TO_CHAR(begin_date, '%A %B %d, %Y %R')
FROM tab1
```

The symbols in the *format_string* parameter in this example have the following meanings. For a complete list of format symbols and their meanings, see the **GL_DATE** and **GL_DATETIME** environment variables in the [Informix Guide to GLS Functionality](#).

Symbol	Meaning
%A	The full weekday name as defined in the locale
%B	The full month name as defined in the locale
%d	The day of the month as a decimal number
%Y	The year as a 4-digit decimal number
%R	The time in 24-hour notation

The result of applying the specified *format_string* to the **begin_date** column is as follows:

```
Wednesday July 23, 1997 18:45
```

TO_DATE Function

In Dynamic Server, the TO_DATE function converts a character string to a DATETIME value. The function evaluates the *char_expression* parameter as a date according to the date format you specify in the *format_string* parameter, and returns the equivalent date.

If the value of the *char_expression* parameter is null, the result of the function is a null value.

If you omit the *format_string* parameter, the TO_DATE function applies the default DATETIME format to the DATETIME value. The default DATETIME format is specified by the **GL_DATETIME** environment variable.

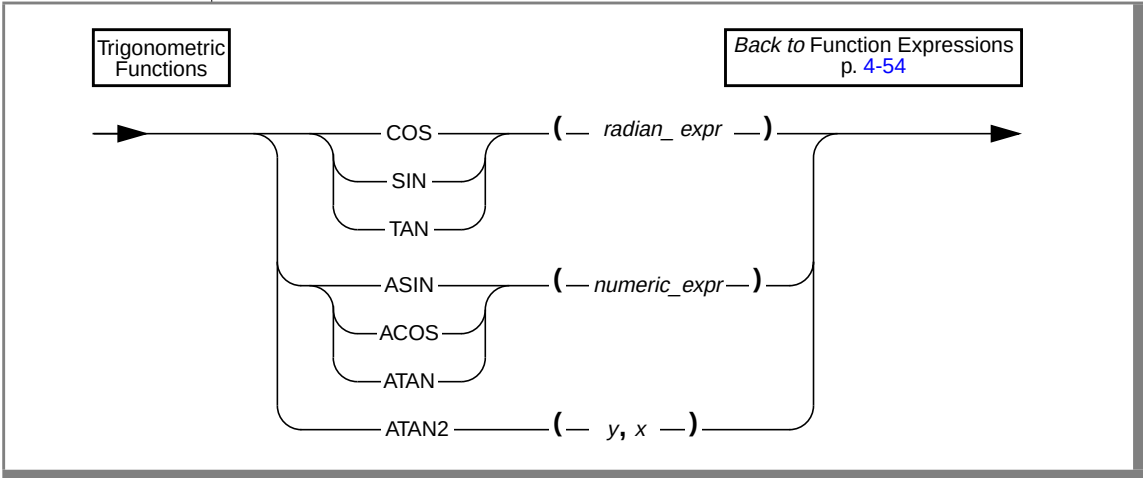
In the following example, the user wants to convert a character string to a DATETIME value in order to update the **begin_date** column of the **tab1** table with the converted value. The **begin_date** column is defined as a DATETIME YEAR TO SECOND data type. The user uses an UPDATE statement that contains a TO_DATE function to accomplish this result.

```
UPDATE tab1
SET begin_date = TO_DATE('Wednesday July 23, 1997 18:45',
'%A %B %d, %Y %R');
```

The *format_string* parameter in this example tells the TO_DATE function how to format the converted character string in the **begin_date** column. For a table that shows the meaning of each format symbol in this format string, see [“TO_CHAR Function” on page 4-78](#).

Trigonometric Functions

A trigonometric function takes an argument, as the following diagram shows.



Element	Purpose	Restrictions	Syntax
<i>numeric_expr</i>	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. 4-33
<i>radian_expr</i>	Expression that evaluates to the number of radians For further information, see “Formulas for Radian Expressions” on page 4-81.	The expression must evaluate to a numeric value.	Expression, p. 4-33
<i>x</i>	Expression that represents the x coordinate of the rectangular coordinate pair (x, y)	The expression must evaluate to a numeric value.	Expression, p. 4-33
<i>y</i>	Expression that represents the y coordinate of the rectangular coordinate pair (x, y)	The expression must evaluate to a numeric value.	Expression, p. 4-33

Formulas for Radian Expressions

The COS, SIN, and TAN functions take the number of radians (*radian_expr*) as an argument.

If you are using degrees and want to convert degrees to radians, use the following formula:

$$\# \text{ degrees} * \pi / 180 = \# \text{ radians}$$

If you are using radians and want to convert radians to degrees, use the following formula:

$$\# \text{ radians} * 180 / \pi = \# \text{ degrees}$$

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  $\theta$  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:

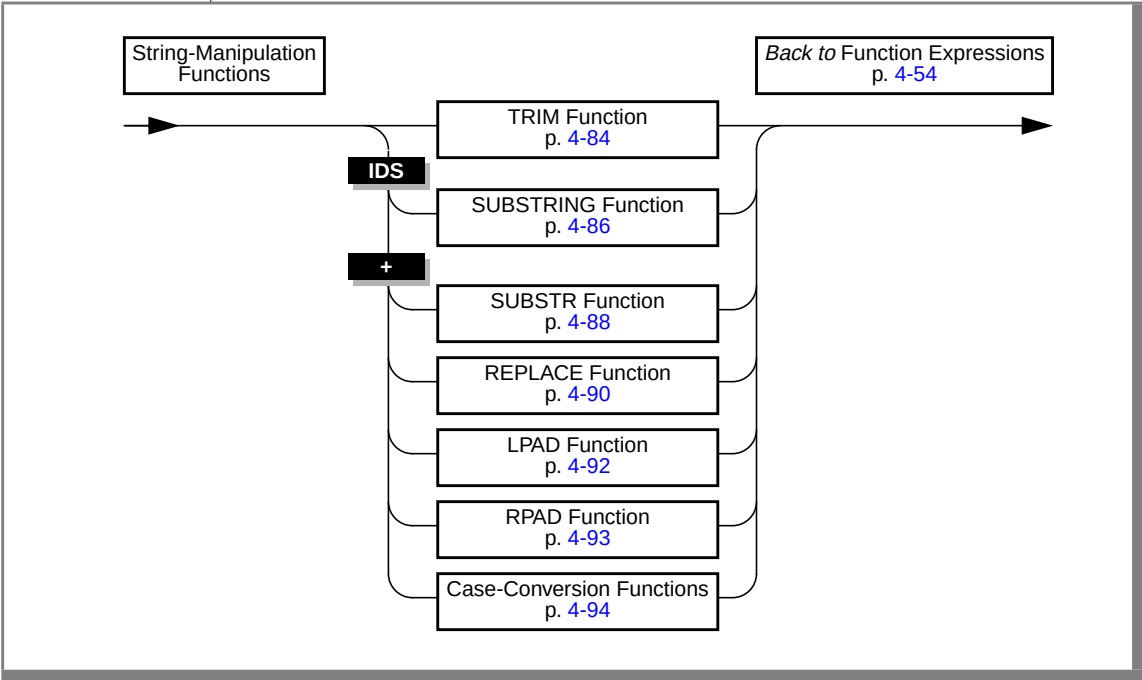
```
SQRT(POW(x,2) + POW(y,2))      --determines  $r$  for (x,y)
```

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)
```

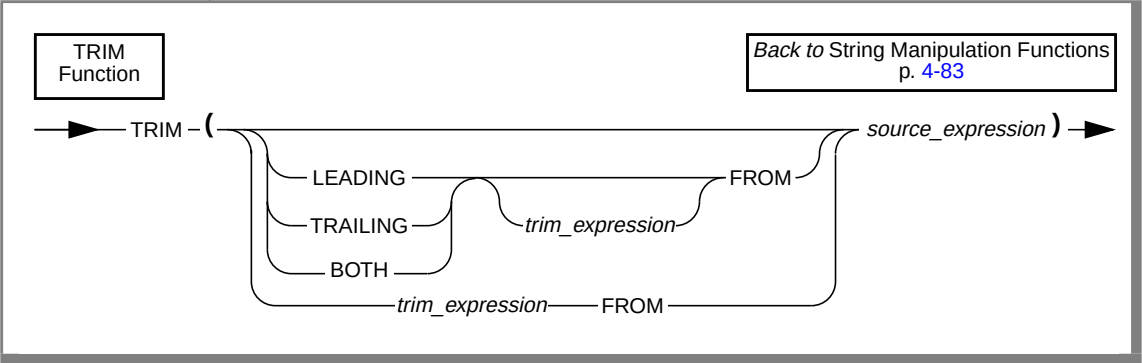
String-Manipulation Functions

String-manipulation functions perform various operations on strings of characters. The syntax for string-manipulation functions is as follows.



TRIM Function

Use the TRIM function to remove leading or trailing (or both) pad characters from a string.



Element	Purpose	Restrictions	Syntax
<i>trim_expression</i>	Expression that evaluates to a single character or null	This expression must be a character expression.	Quoted String, p. 4-157
<i>source_expression</i>	Arbitrary character string expression, including a column or another TRIM function	This expression cannot be a host variable.	Quoted String, p. 4-157

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_expression* is used, a single space is assumed. If either the *trim_expression* or the *source_expression* evaluates to null, the result of the trim function is null. The maximum length of the 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_expression*. For further information on the GLS aspects of the TRIM function in ESQL/C, see the [Informix Guide to GLS Functionality](#). ♦

Fixed Character Columns

The TRIM function can be specified on fixed-length character columns. If the length of the string is not completely filled, the unused characters are padded with blank space. Figure 4-3 shows this concept for the column entry '##A2T##', where the column is defined as CHAR(10).

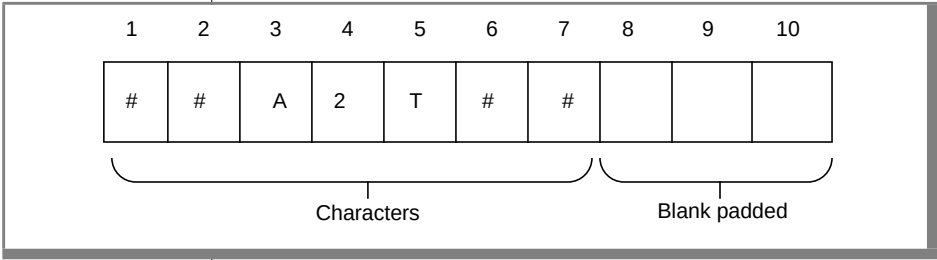


Figure 4-3
Column Entry in a
Fixed-Length
Character Column

If you want to trim the *trim_expression* '#' from the column, you need to consider the blank padded spaces as well as the actual characters. For example, if you specify the trim specification BOTH, the result from the trim operation is A2T##, because the TRIM function does not match the blank padded space that follows the string. In this case, the only '#' trimmed are those that precede the other characters. The SELECT statement is shown, followed by Figure 4-4, which presents the result.

```
SELECT TRIM(BOTH '#' FROM col1) FROM taba
```

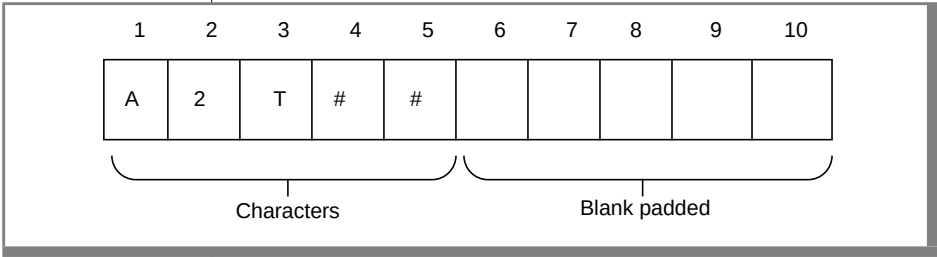


Figure 4-4
Result of TRIM
Operation

The following SELECT statement removes all occurrences of '#':

```
SELECT TRIM(LEADING '#' FROM TRIM(TRAILING ' ' FROM col1)) FROM taba
```

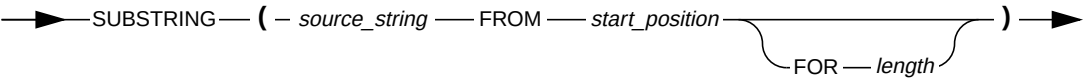
IDS

SUBSTRING Function

In Dynamic Server, the SUBSTRING function returns a subset of a source string.

SUBSTRING
Function

[Back to String Manipulation Functions](#)
[p. 4-83](#)



Element	Purpose	Restrictions	Syntax
<i>length</i>	Number of characters to be returned from <i>source_string</i>	This parameter must be an integer. This parameter can be an expression, constant, column, or host variable.	Literal Number, p. 4-139
<i>source_string</i>	String that serves as input to the SUBSTRING function	This parameter can be any data type that can be converted to a character data type. This parameter can be an expression, constant, column, or host variable.	Expression, p. 4-33
<i>start_position</i>	Column position in <i>source_string</i> where the SUBSTRING function starts to return characters	This parameter must be an integer. This parameter can be an expression, constant, column, or host variable. This parameter can be preceded by a plus sign (+), a minus sign (-), or no sign.	Literal Number, p. 4-139

The subset begins at the column position that *start_position* specifies. The following table shows how the database server determines the starting position of the returned subset based on the input value of the *start_position*.

Value of <i>Start_Position</i>	How the Database Server Determines the Starting Position of the Return Subset
Positive	Counts forward from the first character in <i>source_string</i> For example, if <i>start_position</i> = 1, the first character in the <i>source_string</i> is the first character in the return subset.
Zero (0)	Counts from one position before (that is, left of) the first character in <i>source_string</i> For example, if <i>start_position</i> = 0 and <i>length</i> = 1, the database server returns null, whereas if <i>length</i> = 2, the database server returns the first character in <i>source_string</i> .
Negative	Counts backward from one position before (that is, left of) the first character in <i>source_string</i> For example, if <i>start_position</i> = -1, the starting position of the return subset is two positions (0 and -1) before the first character in <i>source_string</i> .

The size of the subset is specified by *length*. The *length* parameter refers to the number of logical characters rather than to the number of bytes. If you omit the *length* parameter, the SUBSTRING function returns the entire portion of *source_string* that begins at *start_position*.

In the following example, the user specifies that the subset of the source string that begins in column position 3 and is two characters long should be returned.

```
SELECT SUBSTRING('ABCDEFGH' FROM 3 FOR 2)
FROM mytable
```

The following table shows the output of this SELECT statement.

(constant)
CD

In the following example, the user specifies a negative *start_position* for the return subset.

```
SELECT SUBSTRING('ABCDEFG' FROM -3 FOR 7)
FROM mytable
```

The database server starts at the -3 position (four positions before the first character) and counts forward for 7 characters. The following table shows the output of this SELECT statement.

(constant)
ABC

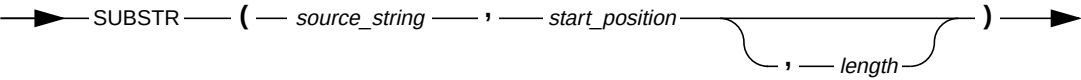
IDS

SUBSTR Function

In Dynamic Server, the SUBSTR function has the same purpose as the SUBSTRING function (to return a subset of a source string), but it uses different syntax.

SUBSTR
Function

Back to String Manipulation Functions
[p. 4-83](#)



Element	Purpose	Restrictions	Syntax
<i>length</i>	Number of characters to be returned from <i>source_string</i>	This parameter must be an integer. This parameter can be an expression, constant, column, or host variable.	Literal Number, p. 4-139
<i>source_string</i>	String that serves as input to the SUBSTR function	This parameter can be any data type that can be converted to a character data type. This parameter can be an expression, constant, column, or host variable.	Expression, p. 4-33
<i>start_position</i>	Column position in <i>source_string</i> where the SUBSTR function starts to return characters	This parameter must be an integer. This parameter can be an expression, constant, column, or host variable. This parameter can be preceded by a plus sign (+), a minus sign (-), or no sign.	Literal Number, p. 4-139

The SUBSTR function returns a subset of *source_string*. The subset begins at the column position that *start_position* specifies. The following table shows how the database server determines the starting position of the returned subset based on the input value of the *start_position*.

Value of <i>Start_Position</i>	How the Database Server Determines the Starting Position of the Return Subset
Positive	Counts forward from the first character in <i>source_string</i>
Zero (0)	Counts forward from the first character in <i>source_string</i> (that is, treats a <i>start_position</i> of 0 as equivalent to 1)
Negative	Counts backward from the last character in <i>source_string</i> A value of -1, returns the last character in <i>source_string</i> .

The *length* parameter specifies the number of characters (not bytes) in the subset. If you omit the *length* parameter, the SUBSTR function returns the entire portion of *source_string* that begins at *start_position*.

In the following example, the user specifies that the subset of the source string to be returned begins at a starting position 3 characters back from the end of the string. Because the source string is 7 characters long, the starting position is the fifth column of *source_string*. Because the user does not specify a value for *length*, the database server returns the entire portion of the source string that begins in column position 5.

```
SELECT SUBSTR('ABCDEFG', -3)
FROM mytable
```

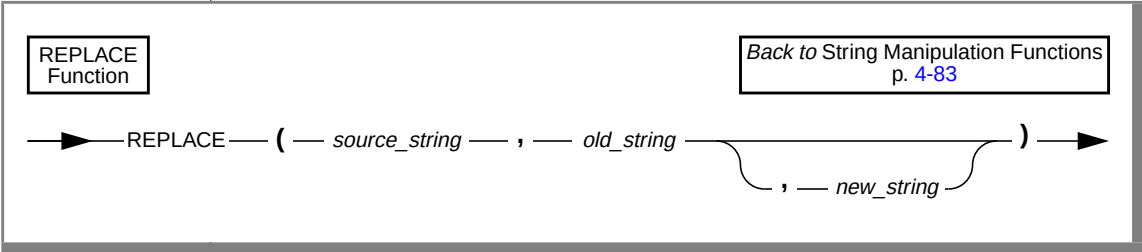
The following table shows the output of this SELECT statement.

(constant)
EFG

IDS

REPLACE Function

In Dynamic Server, the REPLACE function replaces specified characters within a source string with different characters.



Element	Purpose	Restrictions	Syntax
<i>new_string</i>	Character or characters that replace <i>old_string</i> in the return string	This parameter can be any data type that can be converted to a character data type. The parameter can be an expression, column, constant, or host variable.	Expression, p. 4-33
<i>old_string</i>	Character or characters in <i>source_string</i> that are to be replaced by <i>new_string</i>	This parameter can be any data type that can be converted to a character data type. The parameter can be an expression, column, constant, or host variable.	Expression, p. 4-33
<i>source_string</i>	String of characters that serves as input to the REPLACE function	This parameter can be any data type that can be converted to a character data type. The parameter can be an expression, column, constant, or host variable.	Expression, p. 4-33

The REPLACE function returns a copy of *source_string* in which every occurrence of *old_string* is replaced by *new_string*. If you omit the *new_string* option, every occurrence of *old_string* is omitted from the return string.

In the following example, the user replaces every occurrence of xz in the source string with t.

```
SELECT REPLACE('Mighxzy xzime', 'xz', 't')
FROM mytable
```

The following table shows the output of this SELECT statement.

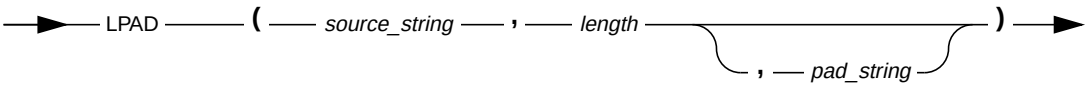
(constant)
Mighty time

LPAD Function

In Dynamic Server, the LPAD function returns a copy of *source_string* that is left-padded to the total number of characters specified by *length*.

LPAD
Function

[Back to String Manipulation Functions](#)
p. 4-83



Element	Purpose	Restrictions	Syntax
<i>length</i>	Integer value that indicates the total number of characters in the return string	This parameter can be an expression, constant, column, or host variable.	Literal Number, p. 4-139
<i>pad_string</i>	String that specifies the pad character or characters	This parameter can be any data type that can be converted to a character data type. The parameter can be an expression, column, constant, or host variable.	Expression, p. 4-33
<i>source_string</i>	String that serves as input to the LPAD function	This parameter can be any data type that can be converted to a character data type. The parameter can be an expression, column, constant, or host variable.	Expression, p. 4-33

The *pad_string* parameter specifies the pad character or characters to be used for padding the source string. The sequence of pad characters occurs as many times as necessary to make the return string reach the length specified by *length*. The sequence of pad characters in *pad_string* is truncated if it is too long to fit into *length*. If you omit the *pad_string* parameter, the default value is a single blank.

In the following example, the user specifies that the source string is to be left-padded to a total length of 16 characters. The user also specifies that the pad characters are a sequence consisting of a dash and an underscore (-_).

```
SELECT LPAD('Here we are', 16, '-_')
FROM mytable
```

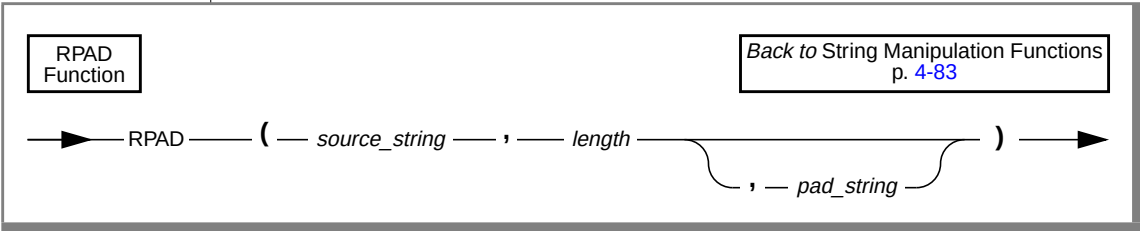
The following table shows the output of this SELECT statement.

(constant)
--_--Here we are

IDS

RPAD Function

In Dynamic Server, the RPAD function returns a copy of *source_string* that is right-padded to the total number of characters that *length* specifies.



Element	Purpose	Restrictions	Syntax
<i>length</i>	Integer value that indicates the total number of characters in the return string	This parameter can be an expression, constant, column, or host variable.	Literal Number, p. 4-139
<i>pad_string</i>	String that specifies the pad character or characters	This parameter can be any data type that can be converted to a character data type. The parameter can be an expression, column, constant, or host variable.	Expression, p. 4-33
<i>source_string</i>	String that serves as input to the RPAD function	This parameter can be any data type that can be converted to a character data type. The parameter can be an expression, column, constant, or host variable.	Expression, p. 4-33

The *pad_string* parameter specifies the pad character or characters to be used to pad the source string. The sequence of pad characters occurs as many times as necessary to make the return string reach the length that *length* specifies. The sequence of pad characters in *pad_string* is truncated if it is too long to fit into *length*. If you omit the *pad_string* parameter, the default value is a single blank.

In the following example, the user specifies that the source string is to be right-padded to a total length of 18 characters. The user also specifies that the pad characters to be used are a sequence consisting of a question mark and an exclamation point (!!)

```
SELECT RPAD('Where are you', 18, '?!')
FROM mytable
```

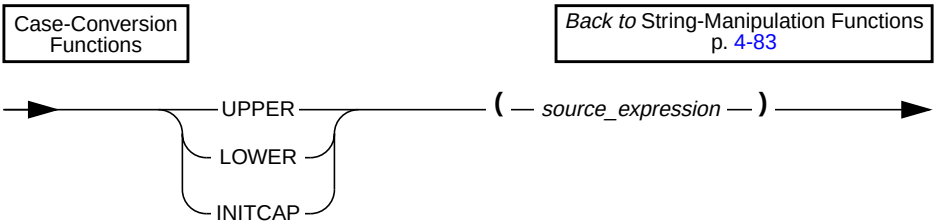
The following table shows the output of this SELECT statement.

(constant)
Where are you?!?!?

IDS

Case-Conversion Functions

In Dynamic Server, case-conversion functions allow you to perform case-insensitive searches on a database and specify the format of the output. The following diagram shows the syntax of case-conversion function expressions.



Element	Purpose	Restrictions	Syntax
<i>source_expression</i>	Column name, function, quoted string, host variable, or any expression that evaluates to a character string	This expression must be a character data type. If you use a host variable, the variable must be declared with a length long enough to handle the converted string.	Expression, p. 4-33

The case-conversion functions perform the following functions:

- ★ UPPER
This function returns a copy of the *source_expression* in which every lowercase alphabetic character in the *source_expression* is replaced by a corresponding uppercase alphabetic character.
- ★ LOWER
This function returns a copy of the *source_expression* in which every uppercase alphabetic character in the *source_expression* is replaced by a corresponding lowercase alphabetic character.
- ★ INITCAP
This function returns a copy of the *source_expression* in which every word in the *source_expression* begins with uppercase letter.
With this function, a word begins after any character other than a letter. Thus, in addition to a blank space, symbols such as commas, periods, colons, and so on, introduce a new word.

The input type must be a character data type. When the column is described, the data type the database server returns is the same as the input type. For example, if the input type is CHAR, the output type is also CHAR.

The byte length returned from the describe of a column with a case-conversion function is the input byte length of the source string. If you use a case-conversion function with a multibyte *source_expression*, the conversion might increase or decrease the length of the string. If the byte length of the result string exceeds the byte length of the *source_expression*, the database server truncates the result string to fit into the byte length of the *source_expression*. ♦

If the *source_expression* is null, the result of a case-conversion function is also null.

The database server treats a case conversion function as a stored procedure in the following instances:

- If it has no argument
- If it has one argument, and that argument is a named argument
- If it has more than one argument
- If it appears in a SELECT list with a host variable as an argument

If none of the conditions in the preceding list are met, the database server treats a case-conversion function as a system function.

Examples

The following example illustrates how you can use case conversion functions to specify the format in which you want values returned.

```
Input Record: SAN Jose
SELECT City, LOWER(City), Lower("City"), UPPER (City),
        INITCAP(City)
FROM Weather;
```

Query output:

```
SAN Jose san jose city SAN JOSE San Jose
```

The following example shows how to use the UPPER function to perform a case-insensitive search on the **lname** column for all employees with the last name of **curran**. Because the INITCAP function is specified in the select list, the database server returns the results in a mixed-case format. For example, the output of one matching row might read: accountant James Curran.

```
SELECT title, INITCAP(fname), INITCAP(lname) FROM employees
WHERE UPPER (lname) = "CURRAN"
```

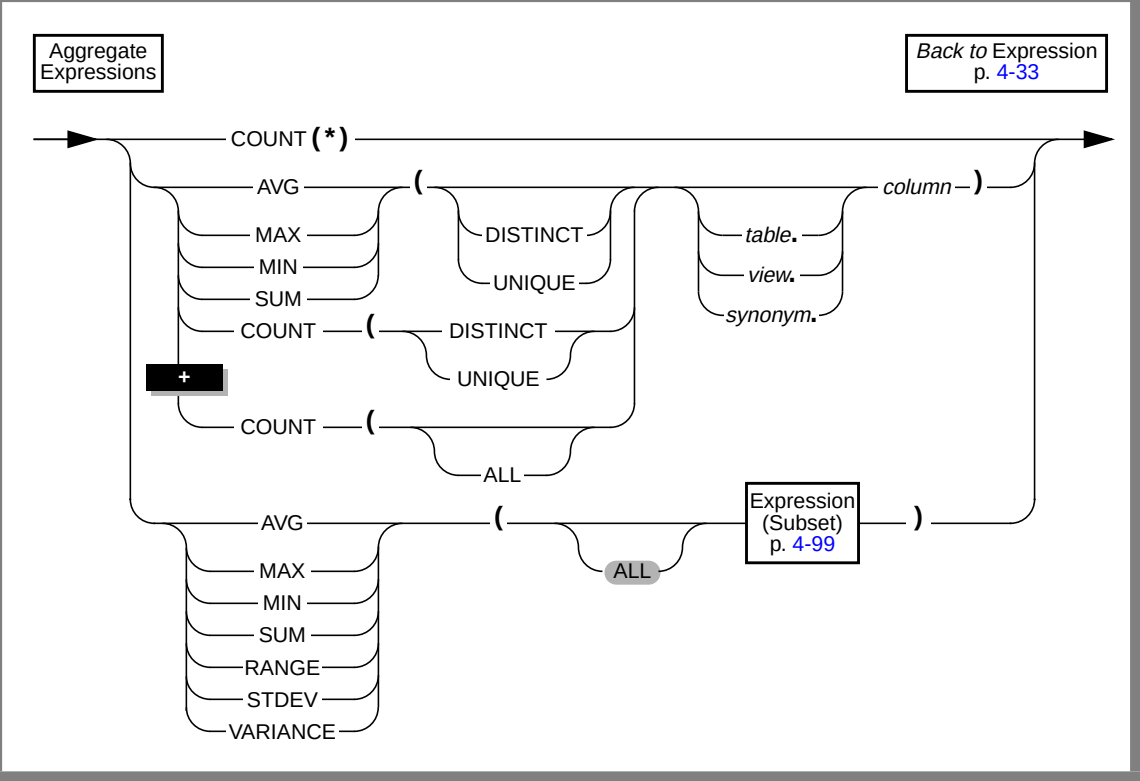
The following example shows how to use the LOWER function to perform a case-insensitive search on the **City** column. This statement directs the database server to replace all instances (that is, any variation) of the words san jose, with the mixed-case format, San Jose.

```
UPDATE Weather SET City =("San Jose")
WHERE LOWER (City = "san jose");
```

Aggregate Expressions

An aggregate expression uses an aggregate function to summarize selected database data.

The following diagram shows the syntax of aggregate function expressions.



Element	Purpose	Restrictions	Syntax
<i>column</i>	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. For other general restrictions, see “ Subset of Expressions Allowed in an Aggregate Expression ” on page 4-99 . For restrictions that depend on the keywords that precede <i>column</i> , see the headings for individual keywords on the following pages.	Identifier, p. 4-113
<i>synonym</i>	Name of the synonym in which the specified column occurs	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table in which the specified column occurs	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view in which the specified column occurs	The view must exist.	Database Object Name, p. 4-25

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

For the full syntax of expressions, see [page 4-33](#).

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. Each form of the COUNT function is explained in the following subsections.

COUNT(*) Keyword

The COUNT (*) keyword 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.

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
```

COUNT column Option

The `COUNT column` 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` option:

```
SELECT COUNT (ALL item_num) FROM items
```

Comparison of the Different Forms of the COUNT Function

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
<code>COUNT (*)</code>	Returns the number of rows that satisfy the query If you do not specify a <code>WHERE</code> clause, this option returns the total number of rows in the table.
<code>COUNT DISTINCT</code> or <code>COUNT UNIQUE</code>	Returns the number of unique non-null values in the specified column
<code>COUNT (column)</code> or <code>COUNT (ALL column)</code>	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](#).

Examples of the Count(*) Option

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. 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

Examples of the COUNT column Option

In the following example the user wants to know how many non-null values are in the **ship_instruct** column of the **orders** table. The user enters the COUNT *column* 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

Examples of the COUNT DISTINCT Option

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. 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

AVG Keyword

The AVG keyword returns the average of all values in the specified column or expression. You can apply the AVG keyword 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 keyword returns a null for that column.

MAX Keyword

The MAX keyword 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 keyword returns a null for that column.

MIN Keyword

The MIN keyword 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 keyword returns a null for that column.

SUM Keyword

The SUM keyword 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 keyword returns a null for that column.

You cannot use the SUM keyword with a character column.

RANGE Keyword

The RANGE keyword 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 Keyword

The STDEV keyword 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 Keyword

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:

$$\frac{(\text{SUM}(X_i^2) - (\text{SUM}(X_i))^2 / N)}{(N-1)}$$

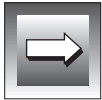
In this example, X_i is each value in the column and N is the total number of values in the column. You can apply the VARIANCE function only to numeric columns. The following query finds the variance on a population:

```
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
3
3
4
(null)

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
2.666666666666667

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	2.666666666666667
AVG (DISTINCT)	3.000000000000000
MAX	4
MAX(DISTINCT)	4
MIN	2
MIN(DISTINCT)	2
SUM	16
SUM(DISTINCT)	9
COUNT(DISTINCT)	3
COUNT(ALL)	6
RANGE	2
STDEV	0.81649658092773
VARIANCE	0.666666666666667

Error Checking in ESQ/LC

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.

Using Arithmetic Operators with Expressions

You can combine expressions with arithmetic operators to make complex expressions. You cannot combine expressions that use aggregate functions with column expressions. The following examples use arithmetic operators:

```
quantity * total_price  
price * 2  
COUNT(*) + 2
```

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:

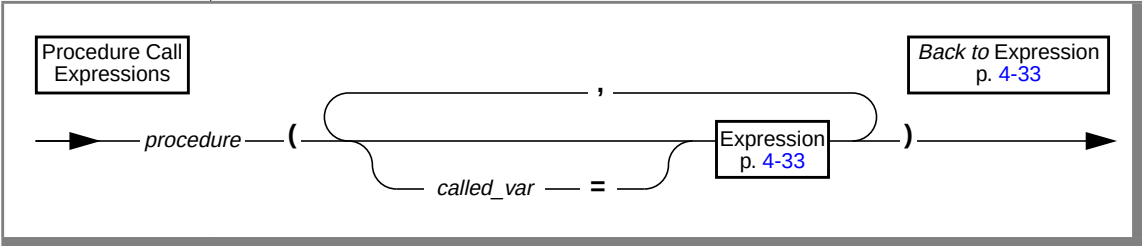
```
SELECT order_num, ship_charge/ship_weight FROM orders  
WHERE order_num = 1023
```

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.

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.

Procedure Call Expressions

The following diagram shows procedure call expressions.



Element	Purpose	Restrictions	Syntax
<i>called_var</i>	Name of a parameter for which you supply an argument to the procedure The parameter name is originally specified in a CREATE PROCEDURE statement, then used in an EXECUTE PROCEDURE statement.	If you use the <i>called_var</i> option for any argument in the called procedure, you must use it for all arguments in the procedure. That is, you must use the <i>called_var = expression</i> syntax for all or none of the arguments in the called procedure.	DEFINE statement, p. 3-8
<i>procedure</i>	Name of the called procedure	The called procedure must exist.	Database Object Name, p. 4-25

Some typical procedure call expressions are shown in the following examples. The first example omits the *called_var* option, and the second example uses the *called_var* option.

```
read_address('Miller')
read_address(lastname = 'Miller')
```

References

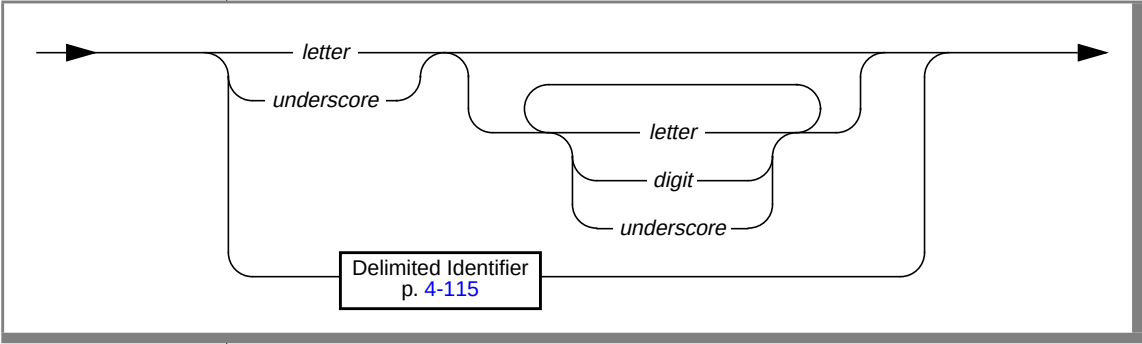
For a discussion of expressions in the context of the SELECT statement, see the [*Informix Guide to SQL: Tutorial*](#).

For discussions of column expressions, length functions, and the TRIM function, see the [*Informix Guide to GLS Functionality*](#).

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
<i>digit</i>	Integer that forms part of the identifier	You must specify a number between 0 and 9, inclusive.	Literal Number, p. 4-139
<i>letter</i>	Letter that forms part of the identifier	If you are using the default locale, a <i>letter</i> 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, <i>letter</i> must be an alphabetic character that the locale supports. For further information, see “ Support for Non-ASCII Characters in Identifiers ” on page 4-115.	Letters are literal values that you enter from the keyboard.
<i>underscore</i>	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.	Underscore character (<code>_</code>) is a literal value that you enter from the keyboard.

Usage

An identifier can contain up to 18 bytes, inclusive.

Use of Uppercase Characters

You can specify the name of a database object with uppercase characters, but the database server shifts the name to lowercase characters unless the **DELIMIDENT** environment variable is set and the name of the database object is enclosed in double quotes. When these conditions are true, the database server treats the name of the database object as a delimited identifier and preserves the uppercase characters in the name. For further information on delimited identifiers, see [“Delimited Identifiers” on page 4-115](#).

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. For a discussion of the syntactic ambiguities that can result from using reserved words as identifiers and an explanation of workarounds for these problems, see [“Potential Ambiguities and Syntax Errors” on page 4-118](#).

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. For the syntax and usage of delimited identifiers, see [“Delimited Identifiers” on page 4-115](#).



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.*

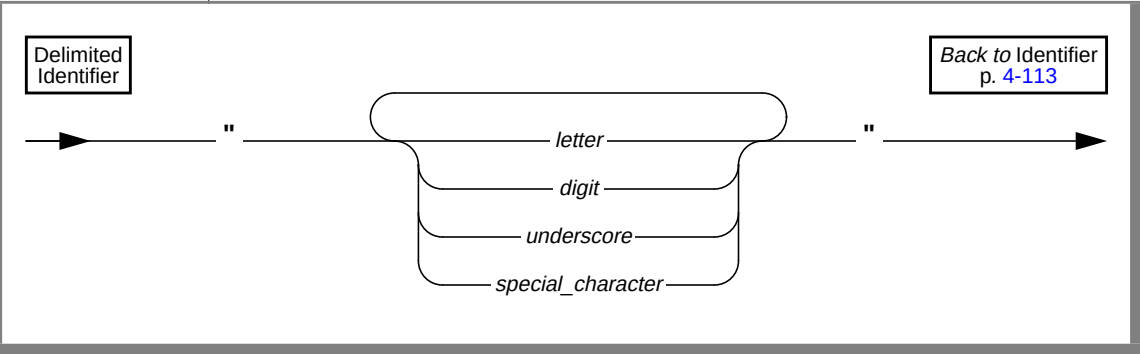
For a list of all the reserved words in the Informix implementation of SQL, see [Appendix A](#).

GLS

Support for Non-ASCII Characters in Identifiers

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 [Informix Guide to GLS Functionality](#).

Delimited Identifiers



Element	Purpose	Restrictions	Syntax
<i>digit</i>	Integer that forms part of the delimited identifier	You must specify a number between 0 and 9, inclusive.	Literal Number, p. 4-139
<i>letter</i>	Letter that forms part of the delimited identifier	Letters in delimited identifiers are case-sensitive. If you are using the default locale, a <i>letter</i> 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, <i>letter</i> must be an alphabetic character that the locale supports. For more information, see “Support for Non-ASCII Characters in Delimited Identifiers” on page 4-117 .	Letters are literal values that you enter from the keyboard.
<i>special_character</i>	Nonalphanumeric character, such as #, \$, or a 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.
<i>underscore</i>	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.	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.

When you create a database object, avoid including one or more trailing blanks in a delimited identifier. In other words, immediately follow the last non-blank character of the name with the end quote.

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

When you are using a nondefault locale whose code set supports non-ASCII characters, you can specify non-ASCII characters in most delimited identifiers. The rule is that if you can specify non-ASCII characters in the undelimited form of the identifier, you can also specify non-ASCII characters in the delimited form of the same identifier. For a list of identifiers that support non-ASCII characters and for information on non-ASCII characters in delimited identifiers, see the [Informix Guide to GLS Functionality](#).

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 enclosed in double quotes (") are treated as identifiers and database objects enclosed in single quotes (') are treated as strings. If the **DELIMIDENT** environment variable is not set, values enclosed in double quotes are also treated as strings.

If 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 enclosed 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 (") in 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','1/22/98')

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 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

In Dynamic Server, 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 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 Procedure Variable Names

If you use any of the following keywords as identifiers for variables in a procedure, you can create ambiguous syntax.

CURRENT	OFF
DATETIME	ON
GLOBAL	PROCEDURE
INTERVAL	SELECT
NULL	

Using CURRENT, DATETIME, INTERVAL, and NULL in INSERT

You cannot use the **CURRENT**, **DATETIME**, **INTERVAL**, or **NULL** keyword as the name of a procedure with the **INSERT** statement.

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 pname 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, pname 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
```

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 LOCK MODE
SET DEBUG FILE	SET LOG
SET EXPLAIN	SET OPTIMIZATION
SET ISOLATION	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

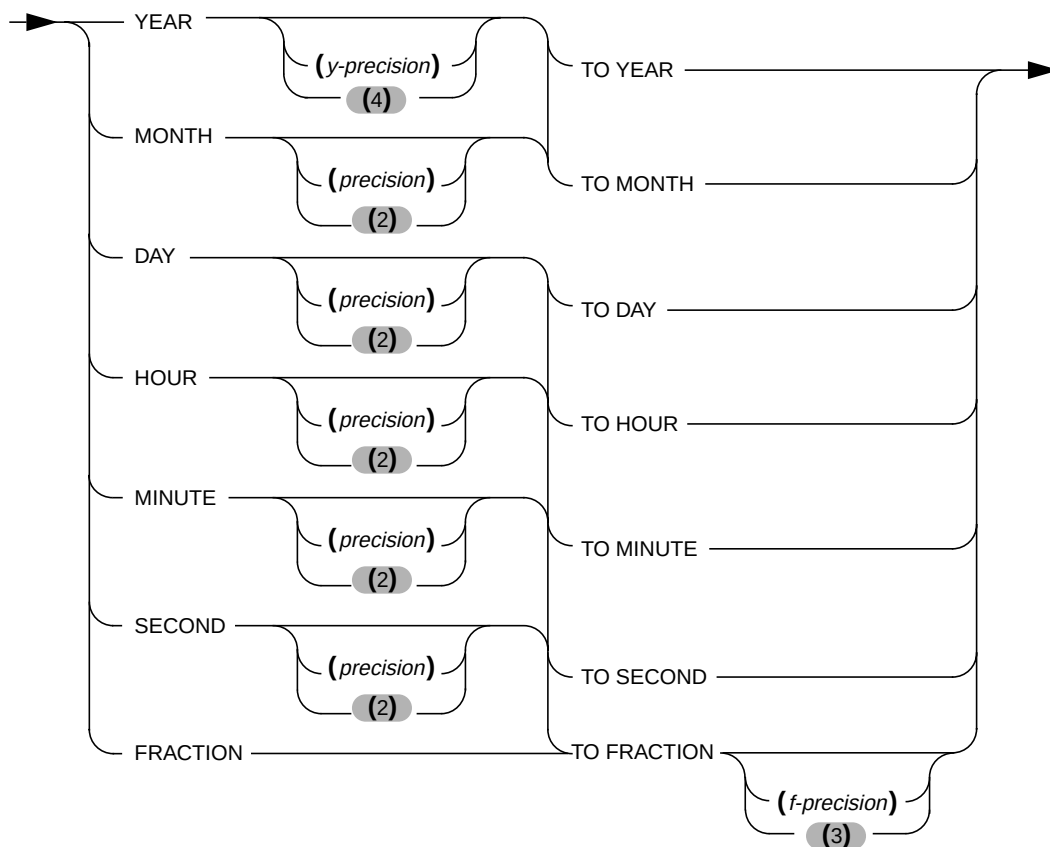
For a discussion of owner naming, see your [Performance Guide](#).

For a discussion of identifiers that support non-ASCII characters and a discussion of non-ASCII characters in delimited identifiers, see the [Informix Guide to GLS Functionality](#).

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
<i>f-precision</i>	Maximum number of digits used in the fraction field The default value of <i>f-precision</i> is 3.	The maximum value that you can specify in <i>f-precision</i> is 5.	Literal Number, p. 4-139
<i>precision</i>	Number of digits in the largest number of months, days, hours, or minutes that the interval can hold The default value of <i>precision</i> is 2.	The maximum value that you can specify in <i>precision</i> is 9.	Literal Number, p. 4-139
<i>y-precision</i>	Number of digits in the largest number of years that the interval can hold The default value of <i>y-precision</i> is 4.	The maximum value that you can specify in <i>y-precision</i> is 9.	Literal Number, p. 4-139

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)
```

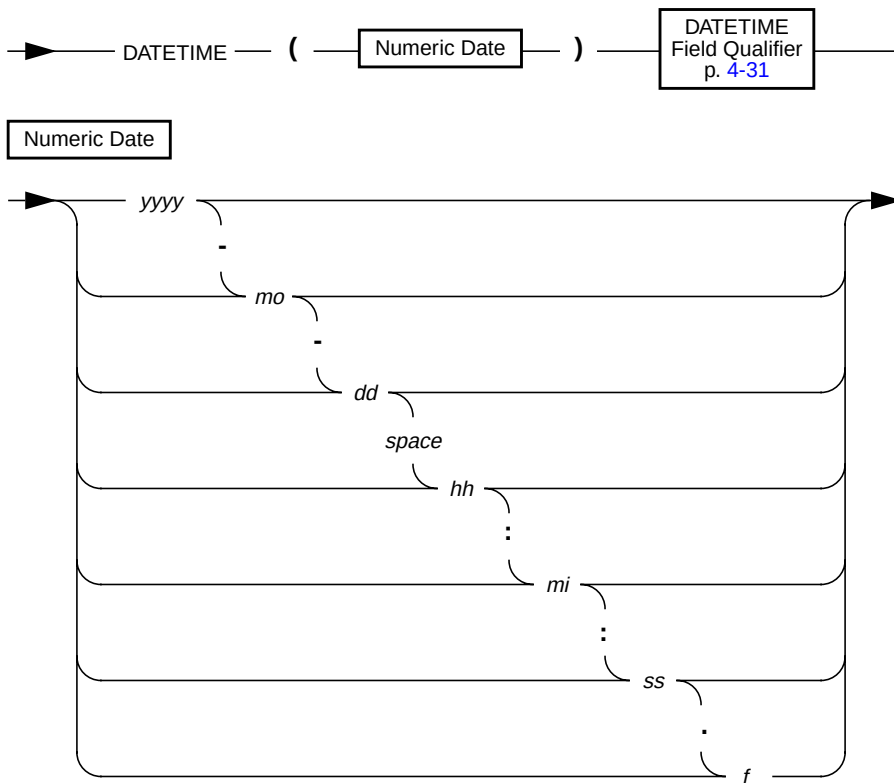
References

For information about how to specify INTERVAL field qualifiers and use INTERVAL data in arithmetic and relational operations, see the discussion of the INTERVAL data type in the [Informix Guide to SQL: Reference](#).

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
<i>dd</i>	Day expressed in digits	You can specify up to 2 digits.	Literal Number, p. 4-139
<i>f</i>	Decimal fraction of a second expressed in digits	You can specify up to 5 digits.	Literal Number, p. 4-139
<i>hh</i>	Hour expressed in digits	You can specify up to 2 digits.	Literal Number, p. 4-139
<i>mi</i>	Minute expressed in digits	You can specify up to 2 digits.	Literal Number, p. 4-139
<i>mo</i>	Month expressed in digits	You can specify up to 2 digits.	Literal Number, p. 4-139
<i>space</i>	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.
<i>ss</i>	Second expressed in digits	You can specify up to 2 digits.	Literal Number, p. 4-139
<i>yyyy</i>	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. 4-139

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 (97-3-6) YEAR TO DAY
```

```
DATETIME (09:55:30.825) HOUR TO FRACTION
```

```
DATETIME (97-5) YEAR TO MONTH
```

The following example shows a literal DATETIME value used with the EXTEND function:

```
EXTEND (DATETIME (1997-8-1) YEAR TO DAY, YEAR TO MINUTE)  
- INTERVAL (720) MINUTE (3) TO MINUTE
```

References

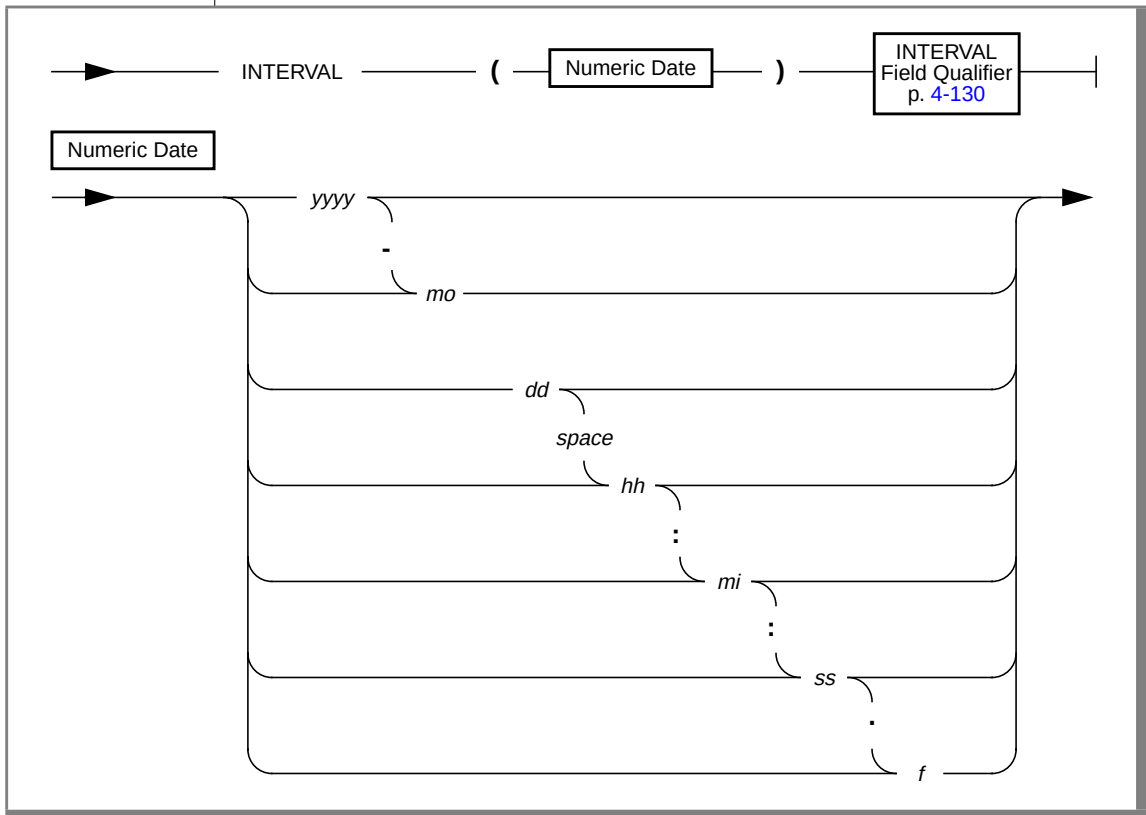
For discussions of the DATETIME data type and the **DBCENTURY** environment variable, see the [Informix Guide to SQL: Reference](#).

For a discussion of customizing DATETIME values for a locale, see the [Informix Guide to GLS Functionality](#).

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
<i>dd</i>	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. 4-139
<i>f</i>	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. 4-139
<i>hh</i>	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. 4-139
<i>mi</i>	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. 4-139
<i>mo</i>	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. 4-139
<i>space</i>	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.
<i>ss</i>	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. 4-139
<i>yyyy</i>	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. 4-139

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
```

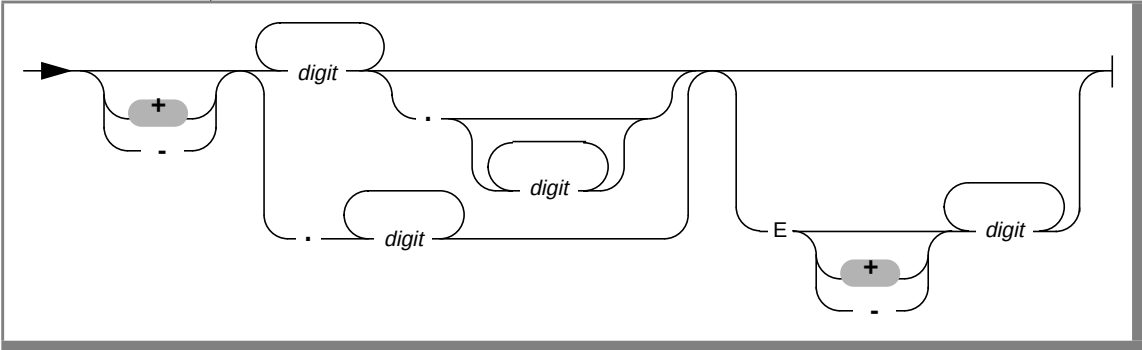
References

For information on how to use INTERVAL data in arithmetic and relational operations, see the discussion of the INTERVAL data type in the [Informix Guide to SQL: Reference](#).

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
<i>digit</i>	Digit that forms part of the literal number	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

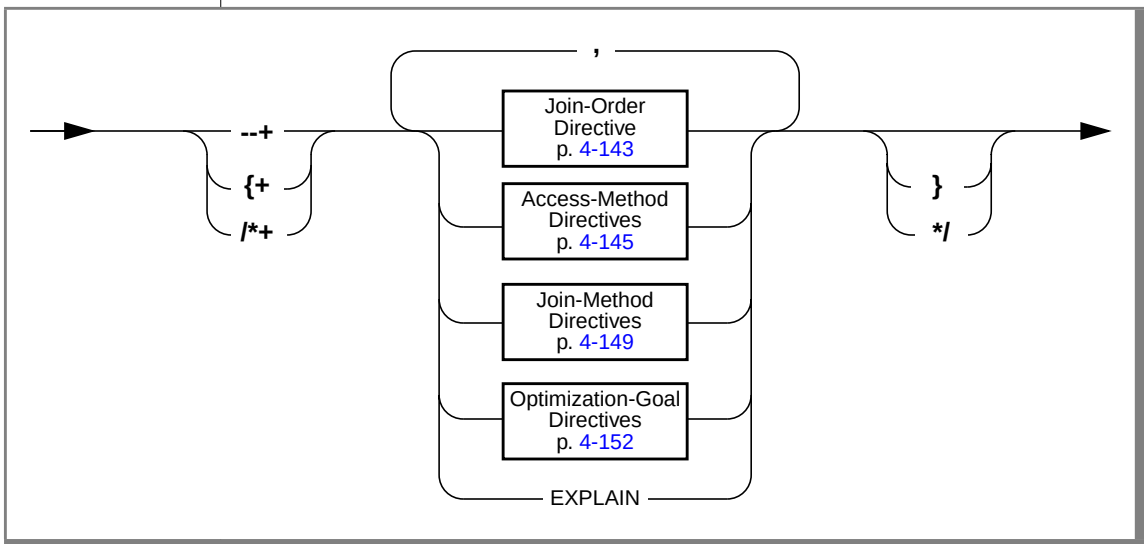
For discussions of numeric data types, such as DECIMAL, FLOAT, INTEGER, and MONEY, see the [*Informix Guide to SQL: Reference*](#).

Optimizer Directives

The Optimizer Directives segment specifies keywords that you can use to partially or fully specify the query plan of the optimizer. Use this segment whenever you see a reference to Optimizer Directives in a syntax diagram.

You can use this segment only with Dynamic Server.

Syntax



Usage

Use one or more optimizer directives to partially or fully specify the query plan of the optimizer.

When you use an optimizer directive, the scope of the optimizer directive is for the current query only.

By default, optimizer directives are enabled. To obtain information about how specified directives are processed, view the output of the SET EXPLAIN statement. To disable optimizer directives, you must set either the **IFX_DIRECTIVES** environment variable to 0 or OFF or the DIRECTIVES parameter in the **ONCONFIG** file to 0.

Optimizer Directives as Comments

An optimizer directive or a string of optimizer directives immediately follows the DELETE, SELECT, or UPDATE keyword in the form of a comment.

After the comment symbol, the first character in a directive is always a plus (+) sign. No space is allowed between the comment symbol and the plus sign.

You can use any of the following comment styles:

- A double dash (--)
The double dash needs no closing symbol because it sets off only one comment line of text.
- Curly brackets ({})
- C-language style comments, slash and asterisk (/**/).

For more information on SQL comment symbols, see [“How to Enter SQL Comments” on page 1-6](#)

If you use multiple directives in one query, you must separate them. You can separate directives with a space, a comma, or any character that you choose. However, Informix recommends that you separate directives with a comma.

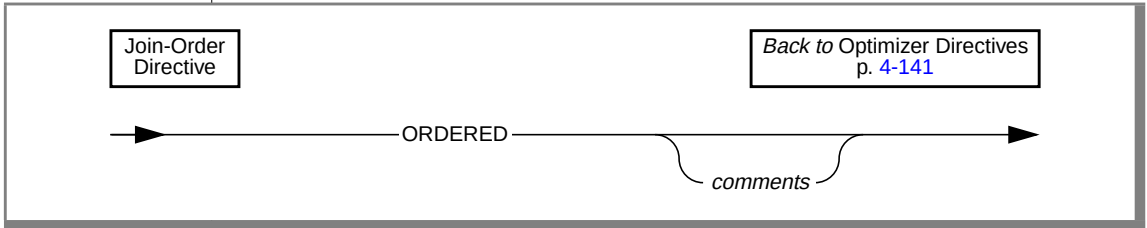
Restrictions on Optimizer Directives

In general, you can specify optimizer directives for any query block in a DELETE, SELECT, or UPDATE statement. However, you cannot use optimizer directives when your statement includes one of the following items:

- Distributed queries, that is, queries that access one or more remote tables
- In ESQL/C, statements that contain the WHERE CURRENT OF cursor clause ♦

Using the Join-Order Directive

Use the ORDERED join-order directive to force the optimizer to join tables in the order in which they appear in the FROM clause.



Element	Purpose	Restrictions	Syntax
<i>comments</i>	Any text that explains the purpose of the directive or other significant information	Text must appear inside the comment symbols.	Character string

For example, the following query forces the database server to join the **dept** and **job** tables, and then join the result with the **emp** table.

```
SELECT --+ ORDERED
       name, title, salary, dname
FROM dept, job, emp
Where title = 'clerk'
AND loc = 'Palo Alto'
AND emp.dno = dept.dno
AND emp.job= job.job
```

Because there are no predicates between the **dept** table and the **job** table, this query forces a Cartesian product.

Using the Ordered Directive with Views

When your query involves a view, the placement of the ORDERED join-order directive determines whether you are specifying a partial or total join order.

- Specifying partial join order when you create a view

If you use the ORDERED join-order directive when you create a view, the base tables are joined contiguously in the order specified in the view definition.

For all subsequent queries on the view, the database server joins the base tables contiguously in the order specified in the view definition. When used in a view, the ORDERED directive does not affect the join order of other tables named in the FROM clause in a query.

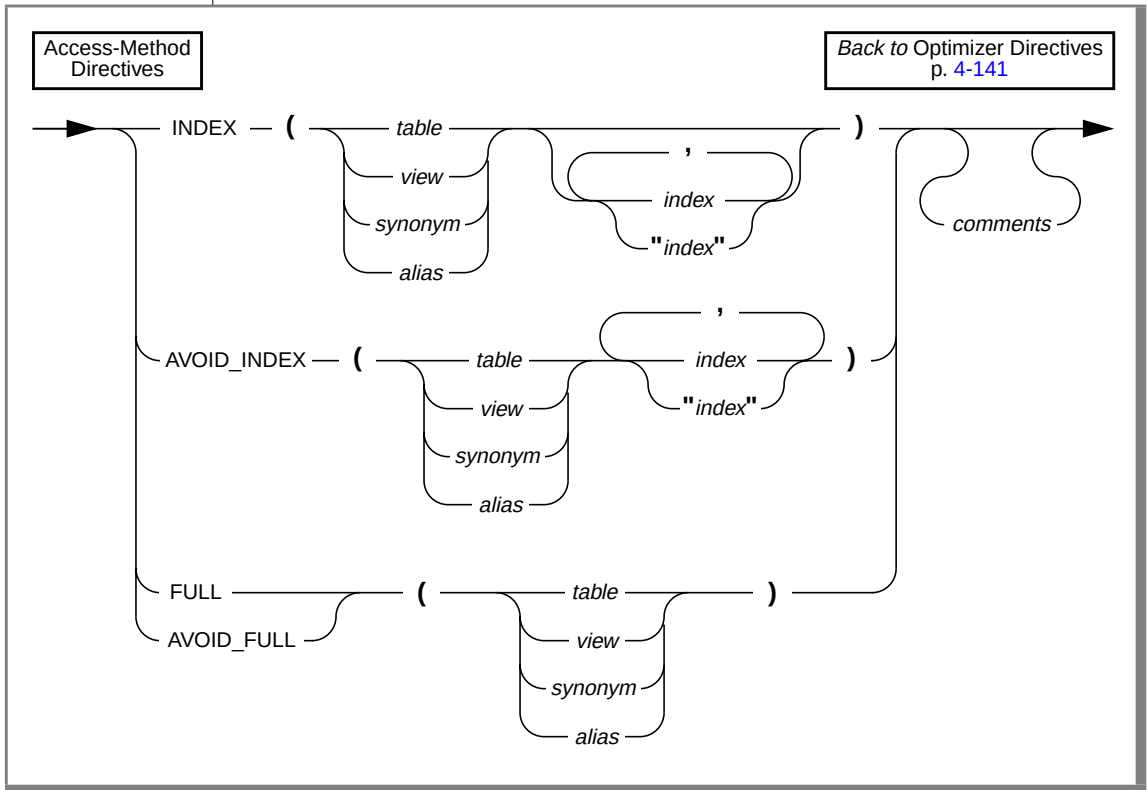
- Specifying total join order when you query a view

When you specify the ORDERED join-order directive in a query that uses a view, all tables are joined in the order specified, even those tables that form views. If a view is included in the query, the base tables are joined contiguously in the order specified in the view definition.

For examples that use the ORDERED join-order directive with views, refer to your [Performance Guide](#).

Access-Method Directives

Use the access-method directive to specify the manner in which the optimizer should search the tables.



Element	Purpose	Restrictions	Syntax
<i>alias</i>	Temporary alternative name assigned to the table or view in the FROM clause	When an alias is declared in the FROM clause, the alias also must be used in the optimizer directive.	Identifier, p. 4-113
<i>comments</i>	Any text that explains the purpose of the directive or other significant information	Text must appear outside the parenthesis, but inside the comment symbols.	Character string
<i>index</i>	Name of the index for which you want to specify a query plan directive	The index must be defined on the specified table. With the AVOID_INDEX directive, at least one index must be specified.	Database Object Name, p. 4-25
<i>synonym</i>	Name of the synonym for which you want to specify a query plan directive	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table for which you want to specify a query plan directive	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view for which you want to specify a query plan directive	The view must exist.	Database Object Name, p. 4-25

You can separate the elements that appear within the parentheses with either one or more spaces or by commas.

The following table lists the purpose of each of the access-method directives and how it affects the query plan of the optimizer.

Keywords	Purpose	Optimizer Action
AVOID_FULL	Do not perform a full-table scan on the listed table.	The optimizer considers the various indexes it can scan. If no index exists, the optimizer performs a full table scan.
AVOID_INDEX	Do not use any of the indexes listed.	The optimizer considers the remaining indexes and a full table scan. If all indexes for a particular table are specified, the optimizer uses a full table scan to access the table.
FULL	Perform a full-table scan.	Even if an index exists on a column, the optimizer uses a full table scan to access the table.
INDEX	Use the index specified to access the table.	If more than one index is specified, the optimizer chooses the index that yields the least cost. If no indexes are specified, then all the available indexes are considered.

Prohibiting a Full Scan of a Table

Both the AVOID_FULL and INDEX keywords specify that the optimizer should avoid a full scan of a table. However, Informix recommends that you use the AVOID_FULL keyword to specify the intent to avoid a full scan on the table. In addition to specifying that the optimizer not use a full-table scan, the negative directive allows the optimizer to use indexes that are created after the access-method directive is specified.

Using Multiple Access-Method Directives on the Same Table

In general, you can specify only one access-method directive per table. However, you can specify both `AVOID_FULL` and `AVOID_INDEX` for the same table. When you specify both these access-method directives, the optimizer avoids performing a full scan of the table and it avoids using the specified index or indexes.

This combination of negative directives allows the optimizer to use indexes that are created after the access-method directives are specified.

Examples that Uses an Access-Method Directive

Suppose that you have a table named **employee**, that contains the following indexes: **loc_no**, **dept_no**, and **job_no**. When you perform a `SELECT` that uses the table in the `FROM` clause you might direct the optimizer to access the table in one of the following ways:

Example Using a Positive Directive

```
SELECT {+INDEX(EMP dept_no)}  
.  
.  
.
```

In this example the access-method directive forces the optimizer to scan the index on the **dept_no** column.

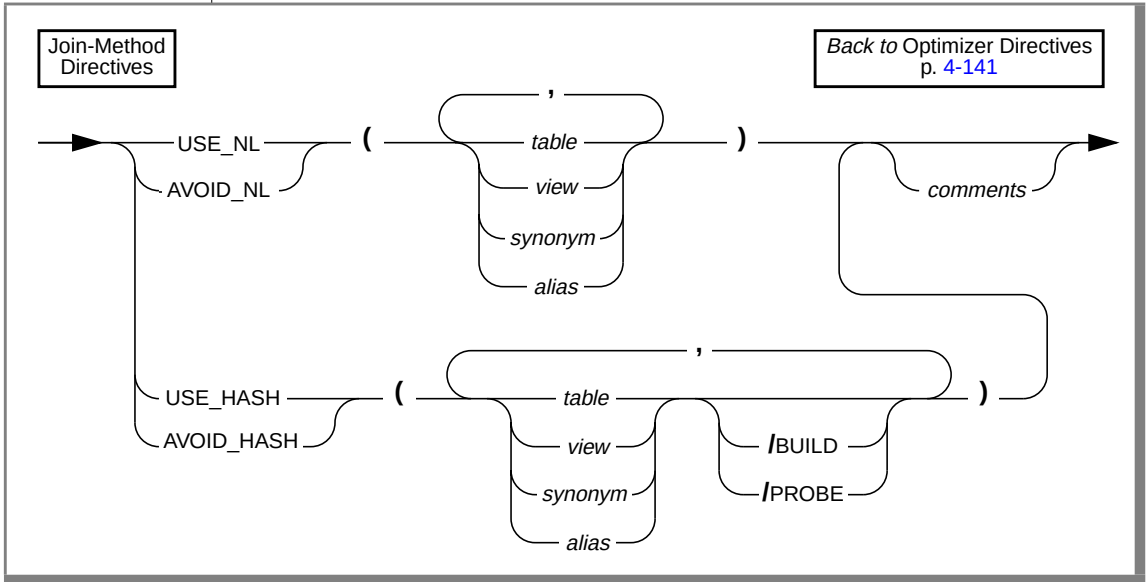
Example Using Negative Directives

```
SELECT {+AVOID_INDEX(EMP loc_no, job_no), AVOID_FULL(EMP)}  
.  
.  
.
```

This example includes multiple access-method directives. These access-method directives also force the optimizer to scan the index on the **dept_no** column. However, if a new index, **emp_no** is created for table **employee**, the optimizer can consider it.

Join-Method Directives

Use join-method directives to influence how the database server joins tables in a query.



Element	Purpose	Restrictions	Syntax
<i>alias</i>	Temporary alternative name assigned to the table or view in the FROM clause	When an alias is declared in the FROM clause, the alias also must be used in the optimizer directive.	Identifier, p. 4-113
<i>comments</i>	Any text that explains the purpose of the directive or other significant information	Text must appear outside the parenthesis, but inside the comment symbols.	Character string
<i>synonym</i>	Name of the synonym for which you want to specify a query plan directive	The synonym and the table to which the synonym points must exist.	Database Object Name, p. 4-25
<i>table</i>	Name of the table for which you want to specify a query plan directive	The table must exist.	Database Object Name, p. 4-25
<i>view</i>	Name of the view for which you want to specify a query plan directive	The view must exist.	Database Object Name, p. 4-25

You can separate the elements that appear within the parentheses with either one or more spaces or by commas.

The following table lists the purpose of each of the join-method directives

Keyword	Purpose
USE_NL	Use the listed tables as the inner table in a nested-loop join. If n tables are specified in the FROM clause, then at most $n-1$ tables can be specified in the USE_NL join-method directive.
USE_HASH	Use a hash join to access the listed table. You can also choose whether the table will be used to create the hash table, or to probe the hash table.
AVOID_NL	Do not use the listed table as the inner table in a nested loop join. A table listed with this directive can still participate in a nested loop join as the outer table.
AVOID_HASH	Do not access the listed table using a hash join. Optionally, you can allow a hash join, but restrict the table from being the one that is probed, or the table from which the hash table is built.

Specifying the Role of the Table in a Hash Join

When you specify that you want to avoid or use a hash join, you can also specify the role of each table:

- BUILD

When used with `USE_HASH`, this keyword indicates that the specified table be used to construct a hash table. When used with `AVOID_HASH`, this keyword indicates that the specified table *not* be used to construct a hash table.

- PROBE

When used with `USE_HASH`, this keyword indicates that the specified table be used to probe the hash table. When used with `AVOID_HASH`, this keyword indicates that the specified table *not* be used to probe the hash table.

If neither the `BUILD` nor `PROBE` keyword is specified, the optimizer uses cost to determine the role of the table.

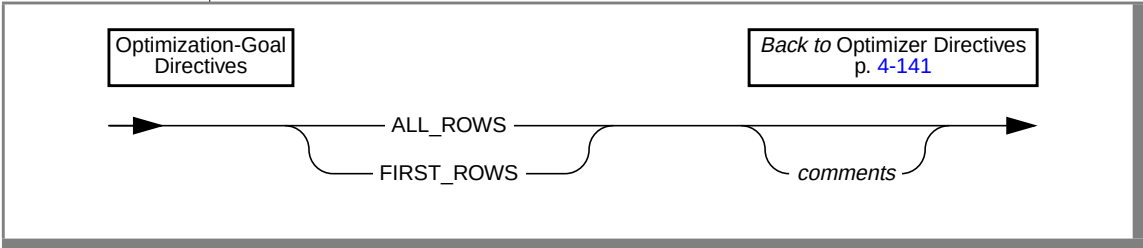
Example Using Join Method Directives

In the following example, the `USE_HASH` join-method directive forces the optimizer to construct a hash join on the **dept** table and consider only the hash join to join the **dept** table with the other tables. Because no other directives are specified, the optimizer can choose the best plan for other joins in the query.

```
SELECT /*+ USE_HASH (dept /BUILD)
        Force the optimizer to use the dept table to
        construct a hash table */
name, title, salary, dname
FROM emp, dept, job
WHERE loc = 'Phoenix'
AND emp.dno = dept.dno
AND emp.job = job.job
```

Optimization-Goal Directives

Use optimization-goal directives to specify the measure that is used to determine the performance of a query result.



Element	Purpose	Restrictions	Syntax
comments	Any text that explains the purpose of the directive or other significant information	Text must appear outside the parenthesis, but inside the comment symbols.	Character string

The two optimization-goal directives are:

- ★ FIRST_ROWS
This directive tells the optimizer to choose a plan that optimizes the process of finding only the first screenful of rows that satisfies the query.
Use this option to decrease initial response time for queries that use an interactive mode or that require the return of only a few rows.
- ★ ALL_ROWS
This directive tells the optimizer to choose a plan that optimizes the process of finding all rows that satisfy the query.
This form of optimization is the default.

Restrictions on Optimization-Goal Directives

You cannot use an optimization-goal directive in the following instances:

- In a view definition
- In a subquery

Example of an Optimization-Goal Directive

The following query returns the names of the employees who earned the top fifty bonuses. The optimization-goal directive directs the optimizer to return the first screenful of rows as fast as possible.

```
SELECT {+FIRST_ROWS
      Return the first screenful of rows as fast as possible}
FIRST 50 lname, fname, bonus
FROM emp
ORDER BY bonus DESC
```

For information about how to set the optimization goal for an entire session, see the SET OPTIMIZATION statement.

Directive-Mode Directive

Use the EXPLAIN directive-mode directive to turn SET EXPLAIN ON for a particular query. You can use this directive to test and debug query plans. Information about the query plan is printed to the **sqexplain.out** file. This directive is redundant when SET EXPLAIN ON is already specified.

You cannot use the EXPLAIN directive-mode directive in two situations:

- In a view definition
- In a subquery

References

For information about the **sqexplain.out** file, see SET EXPLAIN.

For information about how to set optimization settings for an entire session, see SET OPTIMIZATION.

For a discussion about optimizer directives and performance, see your [Performance Guide](#).

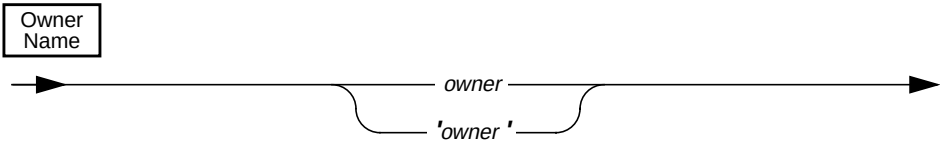
For information on the **IFX_DIRECTIVES** environment variable, see the [Informix Guide to SQL: Reference](#).

For information on the **DIRECTIVES** variable in the **ONCONFIG** file, see your [Administrator's Guide](#).

Owner Name

The owner name segment specifies the name of the owner of a database object in a database. Use this segment whenever you see a reference to Owner Name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
<i>owner</i>	User name of the owner of a database object in a database	If you are using an ANSI-compliant database, you must enter the <i>owner</i> for a database object that you do not own.	Name must conform to the conventions of your operating system.

Usage

In databases that are not ANSI-compliant, the owner name is optional. You do not need to specify *owner* when you use data access statements. However, if you specify *owner*, the database server checks it for correctness. Without quotation marks, *owner* is case insensitive.

The following example shows four queries that can access data successfully from the table **kaths.tab1**:

```
SELECT * FROM tab1
SELECT * FROM kaths.tab1
SELECT * FROM KATHS.tab1
SELECT * FROM KathS.tab1
```

Using Quotation Marks

When you use quotation marks, *owner* is case sensitive. In other words, quotation marks signal the database server to read or store the name exactly as typed. This case sensitivity applies when you create or access a database object.

Suppose you have a table whose owner is **Sam**. You can use one of the following two statements to access data in the table.

```
SELECT * FROM table1
SELECT * FROM 'Sam'.table1
```

The first query succeeds because the owner name is not required. The second query succeeds because the specified owner name matches the owner name as it is stored in the database.

Accessing Information from the System Catalog Tables

If you use the owner name as one of the selection criteria to access database-object information from one of the system catalog tables, the owner name is case sensitive. Because this type of query requires that you use quotation marks, you must type the owner name exactly as it is stored in the system catalog table. Of the following two examples, only the second successfully accesses information on the table **Kaths.table1**.

```
SELECT * FROM systables WHERE tablename = 'table1' AND owner
= 'kaths'
SELECT * FROM systables WHERE tablename = 'table1' AND owner
= 'Kaths'
```

User **informix** is the owner of the system catalog tables.

Tip: The **USER** keyword returns the login name exactly as it is stored on the system. If the owner name is stored differently from the login name (for example, a mixed-case owner name and an all lowercase login name), the `owner = USER` syntax fails.



ANSI

ANSI-Compliant Databases Restrictions and Case Sensitivity

If you specify the owner name when you create or rename a database object in an ANSI-compliant database, you must include the owner name in data access statements. You must include the owner name when you access a database object that you do not own.

The following table describes how the database server reads and stores *owner* when you create, rename, or access a database object.

Owner Name Specification Method	What the Database Server Does
Do not specify	Reads or stores <i>owner</i> exactly as the login name is stored in the system Users must specify <i>owner</i> for a database object or database they do not own.
Specify without quotation marks	Reads or stores <i>owner</i> in uppercase letters
Enclose within quotation marks	Reads or stores <i>owner</i> exactly as typed For more information on how the database server handles this specification method, see “Using Quotation Marks” and “Accessing Information from the System Catalog Tables.”

Because the database server automatically upshifts *owner* if it is not enclosed in quotation marks, case-sensitive errors can cause queries to fail.

For example, 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'
```

The following **SELECT** statement fails because it tries to match the name **NANCY.njcust** to the actual owner and table name of **nancy.njcust**.

```
SELECT * FROM nancy.njcust
```

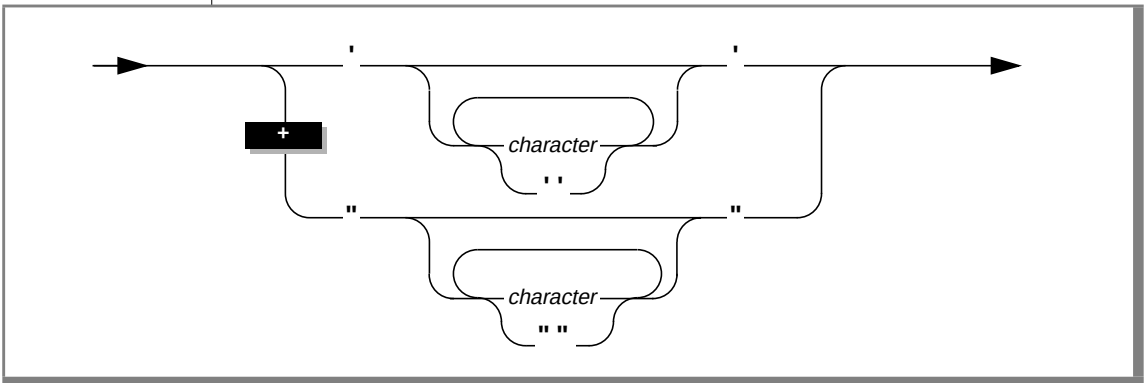


Tip: When you use the owner name as one of the selection criteria in a query, (for example, WHERE owner = 'kaths'), make sure that the quoted string matches the owner name as it is stored in the database. If the database server cannot find the database object or database, you might need to modify the query so that the quoted string uses uppercase letters (for example, WHERE owner = 'KATHS').

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
<i>character</i>	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.	Characters are literal values that you enter from the keyboard.

Usage

The string constant must be written on a single line; that is, you cannot use embedded new lines.

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 in quoted strings, see [“Using Quotes in Strings” on page 4-158](#).
- If you are using a nondefault locale, you can specify non-ASCII characters, including multibyte characters, that the code set of your locale supports. For further information, see the discussion of quoted strings in the [Informix Guide to GLS Functionality](#). ♦
- 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. For further information, see [“Using Quotes in Strings” on page 4-158](#).
- You can enter DATETIME and INTERVAL data as quoted strings. For the restrictions that apply to entering DATETIME and INTERVAL data in quoted-string format, see [“DATETIME and INTERVAL Values as Strings” on page 4-159](#).
- Quoted strings that are used with the LIKE or MATCHES keyword in a search condition can include wildcard characters that have a special meaning in the search condition. For further information, see [“LIKE and MATCHES in a Condition” on page 4-159](#).
- When you insert a value that is a quoted string, you must observe a number of restrictions. For further information, see [“Inserting Values as Quoted Strings” on page 4-160](#).

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. For more information on delimited identifiers, see [“Delimited Identifiers” on page 4-115](#).

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 [page 4-133](#) and [page 4-136](#), 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 ('1997-5-4 10:12:11')
```

```
INSERT INTO manufact(lead_time) VALUES ('14')
```

The format of the value in the quoted string must exactly match the format specified by the qualifiers of the column. For the first case in the preceding example, the **call_dtime** column must be defined with the qualifiers YEAR TO SECOND for the INSERT statement to be valid.

LIKE and MATCHES in a Condition

Quoted strings with the LIKE or MATCHES keyword in a condition can include wildcard characters. For a complete description of how to use wildcard characters, see the [“Condition”](#) segment beginning on [page 4-5](#).

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/yyyy* format or in the format specified by the **DBDATE** environment variable, if set.
- 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

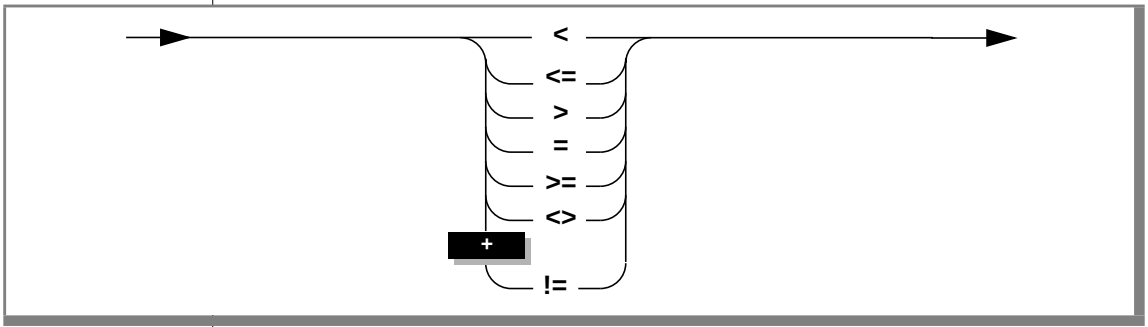
For a discussion of the **DELIMIDENT** environment variable, see the [Informix Guide to SQL: Reference](#).

For a discussion of the GLS aspects of quoted strings, see the [Informix Guide to GLS Functionality](#).

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
$\neq$	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 and VARCHAR 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. For more information on locale-based collation order and the NCHAR and NVARCHAR data types, see the [Informix Guide to GLS Functionality](#). ♦

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.

UNIX

On UNIX, the default code set is the ISO8859-1 code set, which consists of the following sets of characters:

- The ASCII characters have code points in the range of 0 to 127.
This range contains control characters, punctuation symbols, English-language characters, and numerals.
- The 8-bit characters have code points in the range 128 to 255.
This range includes many non-English-language characters (such as é, â, ö, and ñ) and symbols (such as £, ©, and ¿). ♦

WIN NT

On Windows NT, the default code set is Microsoft 1252. This code set includes both the ASCII code set and a set of 8-bit characters. ♦

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	Num	Char	Num	Char	Num	Char	Num	Char
0	^@	20	^T	40	(	60	<	80	P	100	d	120	x
1	^A	21	^U	41	)	61	=	81	Q	101	e	121	y
2	^B	22	^V	42	*	62	>	82	R	102	f	122	z
3	^C	23	^W	43	+	63	?	83	S	103	g	123	{
4	^D	24	^X	44	,	64	@	84	T	104	h	124	
5	^E	25	^Y	45	-	65	A	85	U	105	i	125	}
6	^F	26	^Z	46	.	66	B	86	V	106	j	126	~
7	^G	27	esc	47	/	67	C	87	W	107	k	127	del
8	^H	28	^\ ^]	48	0	68	D	88	X	108	l		
9	^I	29		49	1	69	E	89	Y	109	m		
10	^J	30	^^	50	2	70	F	90	Z	110	n		
11	^K	31	^_ ^_	51	3	71	G	91	[	111	o		
12	^L	32		52	4	72	H	92	\	112	p		
13	^M	33	!	53	5	73	I	93	]	113	q		
14	^N	34	"	54	6	74	J	94	^	114	r		
15	^O	35	#	55	7	75	K	95	_	115	s		
16	^P	36	\$	56	8	76	L	96	`	116	t		
17	^Q	37	%	57	9	77	M	97	a	117	u		
18	^R	38	&	58	:	78	N	98	b	118	v		
19	^S	39	'	59	;	79	O	99	c	119	w		

Support for ASCII Characters in Nondefault Code Sets

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

For a discussion of relational operators in the SELECT statement, see the [Informix Guide to SQL: Tutorial](#).

For a discussion of the GLS aspects of relational operators, see the [Informix Guide to GLS Functionality](#).

Reserved Words

This appendix lists the reserved words in the Informix implementation of SQL. Although you can use almost any word as an SQL identifier, syntactic ambiguities can occur. An ambiguous statement might not produce the desired results. For a discussion of the workarounds for these cases, see the Identifier-segment on [page 4-113](#).

ADD	COMMIT
AFTER	COMMITTED
ALL	CONNECT
ALL_ROWS	CONSTRAINT
ALTER	CONSTRAINTS
AND	CONTINUE
ANSI	COUNT
ANY	CRCOLS
APPEND	CREATE
AS	CURRENT
ASC	CURSOR
AT	DATABASE
ATTACH	DATASKIP
AUDIT	DATE
AUTHORIZATION	DATETIME
AVG	DAY
BEFORE	DBA
BEGIN	DBPASSWORD
BETWEEN	DEBUG
BOTH	DEC
BUFFERED	DECIMAL
BY	DECLARE
BYTE	DECODE
CALL	DEFAULT
CASCADE	DEFERRED
CASE	DEFINE
CHAR	DELETE
CHAR_LENGTH	DESC
CHARACTER	DETACH
CHARACTER_LENGTH	DIRTY
CHECK	DISABLED
CLOSE	DISTINCT
CLUSTER	DISTRIBUTIONS
CLUSTERSIZE	DOCUMENT
COBOL	DOUBLE
COLUMN	DROP

EACH	HAVING
ELIF	HIGH
ELSE	HOLD
ENABLED	HOURL
END	IF
ERROR	IMMEDIATE
ESCAPE	IN
EXCEPTION	INDEX
EXCLUSIVE	INDEXES
EXEC	INDICATOR
EXECUTE	INIT
EXISTS	INITCAP
EXIT	INSERT
EXPLAIN	INT
EXPRESSION	INTEGER
EXTEND	INTERVAL
EXTENT	INTO
FETCH	IS
FILE	ISOLATION
FILLFACTOR	KEY
FILTERING	LABELEQ
FIRST_ROWS	LABELGE
FLOAT	LABELGLB
FOR	LABELGT
FOREACH	LABELLE
FOREIGN	LABELLT
FORTRAN	LABELLUB
FOUND	LABELTOSTRING
FRACTION	LANGUAGE
FRAGMENT	LEADING
FROM	LET
GLOBAL	LEVEL
GO	LIKE
GOTO	LISTING
GRANT	LOCK
GROUP	LOG

ROUND	TEMP
ROW	TEXT
ROWIDS	THEN
ROWS	TIMEOUT
SCHEMA	TO
SECOND	TRACE
SECTION	TRAILING
SELECT	TRANSACTION
SERIAL	TRIGGER
SERIALIZABLE	TRIGGERS
SESSION	TRIM
SET	UNCOMMITTED
SHARE	UNION
SIZE	UNIQUE
SKALL	UNITS
SKINHIBIT	UNLOCK
SKSHOW	UPDATE
SMALLFLOAT	UPPER
SMALLINT	USING
SOME	VALUES
SQL	VARCHAR
SQLCODE	VARIANCE
SQLERROR	VARYING
STABILITY	VIEW
START	VIOLATIONS
STATISTICS	WAIT
STDEV	WHEN
STEP	WHENEVER
STOP	WHERE
STRINGTOLABEL	WHILE
SUBSTR	WITH
SUBSTRING	WITHOUT
SUM	WORK
SYNONYM	WRITE
SYSTEM	YEAR
TABLE	

Index

A

- ABS function 4-56, 4-58
- ABSOLUTE keyword, in
 - FETCH 2-305
- Access control. *See* Privilege.
- ACCESS FOR keywords 2-371
- ACOS function 4-81
- Action clause, in CREATE
 - TRIGGER 2-204, 2-210
- Active set
 - constructing with OPEN 2-396
 - with OPEN 2-395
- ADD CONSTRAINT clause
 - in ALTER TABLE 2-60
- ADD CRCOLS clause
 - in ALTER TABLE 2-40
- ADD keyword
 - in ALTER TABLE 2-42
- ADD ROWIDS clause
 - in ALTER TABLE 2-40
- AFTER keyword
 - in CREATE TRIGGER 2-205
- Aggregate function
 - ALL keyword 4-98
 - AVG function 4-98
 - COUNT function 4-98
 - DISTINCT keyword 4-98
 - GROUP BY 2-482, 2-483
 - in ESQL 4-111
 - in EXISTS subquery 4-17
 - in expressions 2-459
 - in SELECT 2-459
 - MAX function 4-98
 - MIN function 4-98
 - RANGE function 4-106
 - STDEV function 4-107
 - SUM function 4-98
 - summary 4-109
 - UNIQUE keyword 4-98
 - VARIANCE function 4-108
- Algebraic functions
 - ABS function 4-58
 - MOD function 4-59
 - POW function 4-59
 - ROOT function 4-59
 - ROUND function 4-59
 - SQRT function 4-60
 - syntax 4-56
 - TRUNC function 4-61
- Alias
 - for a table in SELECT 2-468
 - use with GROUP BY clause 2-486
- ALL keyword
 - beginning a subquery 2-478
 - in aggregate function 4-98
 - in condition expression 4-18
 - in DISCONNECT 2-270
 - in expression 4-100
 - in GRANT 2-353, 2-356
 - in REVOKE 2-436, 2-438
 - in REVOKE FRAGMENT 2-446
 - in SELECT 2-453, 2-457
 - with UNION operator 2-451, 2-502
- ALLOCATE DESCRIPTOR
 - statement
 - with concatenation operator 4-36
- ALS. *See* Global Language Support.
- ALTER FRAGMENT
 - with generalized-key index 2-22
- ALTER FRAGMENT statement
 - ADD Clause 2-28
 - ATTACH clause 2-11

- attaching with expression 2-17
- attaching with round-robin 2-17
- DETACH clause 2-19
- DROP clause 2-30
- effects on
 - BYTE and TEXT columns 2-18
 - indexes 2-13, 2-18
 - triggers 2-13, 2-19
- how executed 2-10
- hybrid fragmentation 2-20
- MODIFY clause 2-31
- privileges required 2-10
- reverting to non-fragmented 2-25
- syntax 2-8
- use 2-9
- WITH NO CHECK OPTION 2-14

ALTER INDEX statement

- dropping clustered index 2-35
- syntax 2-34

ALTER keyword

- in GRANT 2-344
- in REVOKE 2-436

Alter privilege 2-355

ALTER TABLE

- ADD CRCOLS clause 2-40
- DROP CRCOLS clause 2-40

ALTER TABLE statement

- ADD clause 2-42
- ADD CONSTRAINT clause 2-60
- adding rowids 2-40
- BEFORE option 2-53
- cascading deletes 2-50
- changing column data type 2-56
- changing table lock mode 2-65
- CHECK clause 2-52
- DEFAULT clause 2-43
- DROP clause 2-54
- DROP CONSTRAINT clause 2-63
- DROP ROWIDS clause 2-40
- dropping a column 2-54
- LOCK MODE clause 2-65
- MODIFY clause 2-56
- MODIFY NEXT SIZE clause 2-64
- privilege for 2-344
- reclustering a table 2-35
- REFERENCES clause 2-48
- syntax 2-37

- table-level constraint
 - definition 2-61
 - TYPE options 2-41
- American National Standards Institute. *See* ANSI compliance.
- AND keyword**
 - in Condition segment 4-5, 4-20
 - with BETWEEN keyword 2-474
- ANSI compliance**
 - ansi flag Intro-13, 2-150, 2-156, 2-231
 - icon Intro-11
 - level Intro-20
 - list of SQL statements 1-12
 - table naming 2-429
 - updating rows 2-626
- ANSI-compliant database**
 - creating 2-91
 - database object naming 4-155
 - description of 2-91
 - FOR UPDATE not required in 2-245
 - table privileges 2-156
 - with BEGIN WORK 2-68
- ANY keyword**
 - in condition expression 4-18
 - in SELECT 2-478
- Application**
 - comments in 1-7
 - single-threaded 2-509
 - thread-safe 2-269, 2-509, 2-511
- Arbitrary rule 2-127**
- Arithmetic functions.** *See* Algebraic functions.
- Arithmetic operators 4-34**
- Array, moving rows into with**
 - FETCH 2-307
- AS keyword**
 - in CREATE VIEW 2-230, 2-232
 - in GRANT 2-344, 2-358
 - in SELECT 2-453
 - with display labels 2-461
 - with table aliases 2-469
- ASC keyword**
 - in CREATE INDEX 2-111
 - in SELECT 2-486, 2-489
 - order with nulls 2-490
- ASCII code set 4-164**
- ASIN function 4-81**

- Asterisk (*)**
 - arithmetic operator 4-34
 - as wildcard character 4-13
 - use in SELECT 2-453
- ATAN function 4-81**
- ATAN2 function 4-81**
- Attached index 2-118**
- Autofree feature**
 - in SET AUTOFREE 2-503
- Automatic type conversion.** *See* Data type conversion.
- AVG function 4-98, 4-105**

B

- Backslash (\)**
 - as escape character 4-13
 - as wildcard character 4-12, 4-13
- BEFORE keyword**
 - in ALTER TABLE 2-53
 - in CREATE TRIGGER 2-204
- BEGIN keyword, in CREATE PROCEDURE 2-143**
- BEGIN WORK statement**
 - locking in a transaction 2-67
 - use in ANSI-compliant databases 2-68
- BEGIN-END block, in CREATE PROCEDURE 2-143**
- BETWEEN keyword**
 - in Condition segment 4-6, 4-10
 - in SELECT 2-474
- Boolean expressions 4-5**
- B-tree cleaner list 2-638**
- BUFFERED keyword, in SET LOG 2-564**
- BUFFERED LOG keywords**
 - in CREATE DATABASE 2-90
- BYTE and TEXT data**
 - effect of isolation on retrieval 2-559, 2-590
 - in CREATE TABLE 2-180
- BYTE data type**
 - in LOAD statement 2-388
 - in UNLOAD statement 2-618
 - storage of 2-180
 - syntax 4-27, 4-29
 - with stored procedures 3-9, 3-15

C

Calculated expression, restrictions
with GROUP BY 2-483

CALL keyword, in the
WHENEVER statement 2-652

CALL keyword, in
WHENEVER 2-646

CALL statement, syntax 3-4

Caret (^)
as wildcard character 4-13
use with square brackets 4-13

CASCADE keyword, in
REVOKE 2-434

Cascading deletes
defined 2-169
definition of 2-50
locking associated with 2-51,
2-170

logging 2-51, 2-170
restrictions on 2-51, 2-170
syntax 2-50, 2-164

Cascading triggers
and triggering table 2-220, 2-225
description of 2-224
maximum number of 2-224
scope of correlation names 2-214
triggered actions 2-206

CASE expression 4-40

Case-conversion functions 4-95

cdrserver, replication column
name 2-178

cdrtime, replication column
name 2-178

CHAR data type
in INSERT 4-161
syntax 4-27
using as default value 2-161

CHARACTER VARYING data type
See VARCHAR data type.
syntax 4-30

CHARACTER_LENGTH
function 4-72

CHECK clause
in ALTER TABLE 2-52
in CREATE EXTERNAL
TABLE 2-97
in CREATE TABLE 2-171

Check constraints
description of 2-97, 2-171
reject files 2-103
unloading data 2-103

CLOSE DATABASE statement
prerequisites to close 2-73
syntax 2-73

CLOSE statement
closing a select cursor 2-71
closing an insert cursor 2-71
cursors affected by transaction
end 2-72
syntax 2-70
with concatenation operator 4-36

CLUSTER keyword
in ALTER INDEX 2-34
in CREATE INDEX 2-109

Clustered index
with ALTER INDEX 2-34
with CREATE INDEX 2-109

CODESET keyword
in CREATE EXTERNAL
TABLE 2-101, 2-500

Collation, with relational
operators 4-163

Column
adding a constraint 2-157
creating 2-42, 2-159
defining as foreign key 2-177
defining as primary key 2-176
dropping 2-54
inserting into 2-375
modifying with ALTER
TABLE 2-56
number allowed when defining
constraint 2-157
number, effect on triggers 2-203
referenced and referencing 2-49,
2-166
renaming 2-426
specifying a subscript 2-488, 4-38
specifying check constraint
for 2-97, 2-171
virtual 2-232

Column expression
in SELECT 2-458
syntax 4-36

Column name
using functions as names 4-119
using keywords as names 4-121
when qualified 2-213

Column value
in triggered action 2-215
qualified vs. unqualified 2-216
when unqualified 2-215

Command file
comment symbols in 1-7
defined 1-7

Comment icons Intro-9

Comment symbol
curly brackets ({}) 1-6
double dash (--) 1-6
examples of 1-7
how to enter 1-6
in application programs 1-7
in command file 1-7
in prepared statements 2-407
in SQL APIs 1-7
in stored procedure 1-7

COMMIT WORK statement
syntax 2-75
use in ANSI-compliant
databases 2-76
use in non-ANSI databases 2-76
Committed Read isolation
level 2-557

COMMITTED READ keywords,
syntax in SET
ISOLATION 2-556

Comparison condition, syntax and
use 4-6

Compliance
icons Intro-11

Compliance, with industry
standards Intro-20

Compound assignment 3-32

Concatenation operator (||) 4-34,
4-35

Concurrency
Committed Read isolation 2-557
Cursor Stability isolation 2-558
Dirty Read isolation 2-557
Read Committed isolation 2-588
Read Uncommitted
isolation 2-588

- Repeatable Read isolation 2-558, 2-588
- Serializable isolation level 2-588
- with SET ISOLATION 2-556
- with SET TRANSACTION 2-589
- Condition segment
 - boolean expressions 4-6
 - comparison condition 4-6
 - description of 4-5
 - join conditions 2-479
 - keywords
 - ALL 4-18
 - ANY 4-18
 - BETWEEN 4-10
 - EXISTS 4-17
 - IN 4-11
 - IS NOT NULL 4-11
 - IS NULL 4-11
 - MATCHES 4-12
 - NOT 4-12
 - SOME 4-18
 - null values 4-6
 - relational operators in 4-10
 - subquery in SELECT 4-15
 - syntax 4-5
 - use of functions in 4-6
 - wildcards in searches 4-12
 - with ESCAPE clause 4-14
- CONNECT keyword
 - in GRANT 2-347
 - in REVOKE 2-440
- Connect privilege 2-347
 - revoking 2-440
- CONNECT statement
 - and INFORMIXSERVER
 - environment variable 2-79
 - connection context 2-79
 - connection identifiers 2-79
 - database environment 2-82
 - DEFAULT option 2-79
 - implicit connections 2-80
 - Locating a database with
 - DBPATH 2-85
 - syntax 2-77
 - use 2-77
 - USER clause 2-86
 - WITH CONCURRENT
 - TRANSACTION option 2-81

- Connection
 - context 2-79, 2-268, 2-508
 - current 2-78, 2-269, 2-511
 - default 2-80, 2-268, 2-511
 - dormant 2-78, 2-268, 2-507
 - identifiers 2-79
 - implicit 2-80, 2-268, 2-511
- Constant expression
 - in SELECT 2-458
 - inserting with PUT 2-420
 - restrictions with GROUP
 - BY 2-483
 - syntax 4-48
- Constraint
 - adding with ALTER TABLE 2-59, 2-60, 2-61, 2-525
 - affected by dropping a column
 - from table 2-54
 - changing with ALTER
 - TABLE 2-158
 - checking 2-225
 - definition of 2-156
 - dropping with ALTER
 - TABLE 2-63
 - enforcing 2-158
 - modifying a column that has
 - constraints 2-56
 - name, specifying 4-25
 - number of columns
 - allowed 2-157
 - privileges needed to create 2-62
 - setting 2-591
 - transaction mode 2-591
 - with DROP INDEX 2-273
- Constraint Name. *See* Database Object Name.
- Consumed table 2-12
- CONTINUE keyword, in the
 - WHENEVER statement 2-646, 2-650
- CONTINUE statement
 - syntax 3-7
- Conventions, for naming
 - tables 2-156
- Correlated subquery, definition
 - of 4-15

- Correlation name
 - and stored procedures 2-214
 - in COUNT DISTINCT
 - clause 2-214
 - in GROUP BY clause 2-214
 - in SET clause 2-214
 - in stored procedure 2-222
 - new 2-210
 - old 2-210
 - rules for 2-214
 - scope of 2-214
 - table of values 2-216
 - using 2-214
 - when to use 2-214
- COS function 4-81
- COUNT DISTINCT clause 2-214
- COUNT field
 - in GET DESCRIPTOR 2-321
 - in WHERE clause 2-545
- COUNT function
 - in SET DESCRIPTOR 2-548
 - use in expression 4-98, 4-100, 4-101
- CRCOLS keyword, in CREATE
 - TABLE 2-178
- CREATE DATABASE statement
 - ANSI compliance 2-91
 - syntax 2-89
 - using with PREPARE 2-89
- CREATE EXTERNAL TABLE
 - statement
 - CHECK clause 2-97
 - column-level constraints 2-96
 - DATAFILES clause 2-98
 - optional item syntax 2-99
 - SAMEAS clause 2-94
 - setting columns NOT NULL 2-96
 - syntax 2-92
- CREATE INDEX statement
 - ASC keyword 2-111
 - CLUSTER keyword 2-109
 - cluster with fragments 2-109
 - composite indexes 2-110
 - DESC keyword 2-111
 - disabled indexes 2-132
 - DISTINCT keyword 2-108
 - FILLFACTOR clause 2-123
 - FRAGMENT BY EXPRESSION
 - clause 2-125

- generalized-key index 2-119
- implicit table locks 2-107
- IN dbspace clause 2-124
- index-type options 2-108
- sort order 2-111
- specifying object modes 2-129
- storage options 2-122
- syntax 2-106
- UNIQUE keyword 2-108
- USING BITMAP keywords 2-107
- CREATE PROCEDURE FROM statement
 - syntax and use 2-144
 - with concatenation operator 4-36
- CREATE PROCEDURE statement
 - BEGIN-END block 2-143
 - syntax 2-134
- CREATE ROLE statement,
 - syntax 2-146
- CREATE SCHEMA statement
 - defining a trigger 2-199
 - specifying owner of created objects with GRANT 2-149
 - syntax 2-148
 - with CREATE sequences 2-149
- CREATE SYNONYM statement
 - chaining synonyms 2-153
 - syntax 2-151
- CREATE TABLE statement
 - cascading deletes 2-169
 - CHECK clause 2-171
 - column-level constraints 2-163
 - DEFAULT clause 2-160
 - default privileges 2-156
 - defining constraints 2-163, 2-175
 - FRAGMENT BY clause 2-181
 - fragmentation by expression with 2-184
 - HYBRID clause 2-187
 - IN dbspace clause 2-179, 2-183
 - LOCK MODE clause 2-188
 - ON DELETE CASCADE keywords 2-164
 - REFERENCES clause 2-164
 - rules for constraints 2-166
 - setting columns NOT NULL 2-164

- specifying extent size 2-189
- specifying table columns 2-159
- storing database tables 2-179
- syntax 2-155
- table types 2-177
- with BYTE and TEXT data types 2-164
- WITH CRCOLS keywords 2-178
- WITH ROWIDS clause 2-182
- CREATE TEMP TABLE statement.
 - See CREATE TABLE statement.
- CREATE TRIGGER statement
 - disabling triggers 2-228
 - enabling triggers 2-228
 - in ESQL/C application 2-199
 - privilege to use 2-199
 - syntax 2-198
 - triggered action clause 2-211
 - use 2-199
- CREATE VIEW statement
 - column data types 2-231
 - privileges 2-231
 - restrictions on 2-231
 - syntax 2-230
 - virtual column 2-232
 - WITH CHECK OPTION 2-234
 - with SELECT * notation 2-231
- Curly brackets ({}) comment
 - symbol 1-6
- Current database
 - specifying with DATABASE 2-236
- CURRENT function
 - in ALTER TABLE 2-43
 - in Condition segment 4-6
 - in expression 4-48, 4-52
 - in INSERT 2-378, 2-380
 - in WHERE condition 4-53
 - input for DAY function 4-53
 - using as default value 2-162
- CURRENT keyword
 - in DISCONNECT 2-269
 - in FETCH 2-305
 - in SET CONNECTION 2-511
- CURRENT OF keywords
 - in DELETE 2-258, 2-261
 - in UPDATE 2-623, 2-633

- Cursor
 - activating with OPEN 2-394
 - affected by transaction end 2-72
 - associating with prepared statements 2-254
 - characteristics 2-246
 - closing 2-70
 - closing with ROLLBACK WORK 2-449
 - declaring 2-241
 - for update 2-245, 2-249
 - restricted statements 2-252
 - using in ANSI-mode databases 2-253
 - using in non-ANSI databases 2-253
 - freeing automatically with SET AUTOFREE 2-503
 - manipulation statements 1-10
 - opening 2-395, 2-396
 - optimization statements 1-10
 - read-only 2-245, 2-251
 - restricted statements 2-252
 - using in ANSI-mode databases 2-253
 - using in non-ANSI databases 2-253
 - retrieving values with FETCH 2-301
 - scroll 2-246
 - sequence of program operations 2-244
 - sequential 2-246
 - statement, as trigger event 2-201
 - with INTO keyword in SELECT 2-464
 - with prepared statements 2-244
 - with transactions 2-255
- Cursor Stability isolation
 - level 2-558
- CURSOR STABILITY keywords, in SET ISOLATION 2-556
- Cursory procedure 2-138, 3-25

D

Data

- access statements 1-10
- definition statements 1-9
- inserting with the LOAD 2-384
- integrity statements 1-11
- manipulation statements 1-10

Data distributions

- confidence 2-644
- on temporary tables 2-644
- RESOLUTION 2-641, 2-644

DATA field, setting with SET DESCRIPTOR 2-551

Data replication 2-40

Data types

- See also* each data type listed under its own name.
- changing with ALTER TABLE 2-58
- considerations for INSERT 2-379, 4-161
- requirements for referential constraints 2-49, 2-167
- specifying with CREATE VIEW 2-231
- syntax 4-27

Database

- closing with CLOSE DATABASE 2-73
- creating with CREATE DATABASE 2-89
- default isolation levels 2-559, 2-589
- dropping 2-271
- lock 2-237
- naming conventions 4-23
- naming with variable 4-24
- opening in exclusive mode 2-237
- optimizing queries 2-637
- remote 4-23
- renaming 2-428

Database Administrator (DBA) 2-348

Database Name segment

- for remote database 4-23
- naming conventions 4-22
- naming with variable 4-24
- syntax 4-22

Database object

- naming 4-25
- naming owner 4-155

Database object mode

- disabled
 - benefits of 2-532
 - defined 2-520
- enabled
 - benefits of 2-533
 - defined 2-520
- examples 2-521, 2-527

filtering

- benefits of 2-533
- defined 2-520
- error options 2-517
- for triggers 2-228
- in SET Database Object Mode statement 2-516
- privileges required 2-513
- use

- with data definition statements 2-525
- with data manipulation statements 2-519

Database object name segment 4-25

DATABASE statement

- determining database type 2-237
- exclusive mode 2-237
- specifying current database 2-236
- SQLWARN after 2-237
- syntax 2-236

Database-level privilege

- description of 2-346
- granting 2-346
- passing grant ability 2-357
- revoking 2-440
- See also* Privilege.

DATAFILES clause

- in CREATE EXTERNAL TABLE 2-97

Dataskip

- skipping unavailable dbspaces 2-535

DATE data type

- functions in 4-73
- syntax 4-27
- using as a default value 2-161

DATE function

- syntax in expression 4-73
- use in expression 4-75

DATETIME data type

- as quoted string 4-160
- field qualifiers 4-32
- in expression 4-53
- in INSERT 4-161
- syntax 4-27, 4-134
- using as default value 2-161

DATETIME Field Qualifier

- segment 4-32

DAY function 4-73, 4-76

DAY keyword

- in DATETIME 4-32
- in DATETIME segment 4-134
- in INTERVAL 4-131, 4-137

DBA keyword, in REVOKE 2-440

DBA permission, with CREATE SCHEMA 2-149

DBANSIWARN environment variable 2-150, 2-156, 2-231

DBA-privileged procedure 2-135

DBCENTURY environment

- variable 2-386

DBDATE environment

- variable 2-386, 2-617

DBDELIMITER environment

- variable 2-389

DBINFO function 4-61

DBMONEY environment

- variable 2-386, 2-617

DBMS. *See* Database management system.

DBPATH environment

- variable 2-85

DBSERVERNAME function

- as default value 2-162
- in ALTER TABLE 2-43
- returning server name 4-51

Dbspace

- setting in CREATE TABLE 2-179, 2-183
 - skipping if unavailable 2-535
- ### DBSPACETEMP environment variable 2-193
- ### DBTIME environment variable 2-387, 2-618

- DDL statements, summary 1-9
- Deadlock detection 2-563
- DEALLOCATE DESCRIPTOR statement
 - syntax 2-239
 - with concatenation operator 4-36
- DECIMAL data type
 - syntax 4-27
 - using as default value 2-161
- DECLARE statement
 - cursor characteristics 2-246
 - cursors with prepared statements 2-254
 - cursors with transactions 2-255
 - FOR READ ONLY
 - keywords 2-245, 2-251
 - FOR UPDATE keywords 2-245
 - hold cursor 2-247
 - insert cursor 2-245, 2-256, 2-257
 - procedure cursor 2-244
 - read-only cursor 2-245, 2-251
 - restrictions with SELECT with ORDER BY 2-491
 - scroll cursor 2-246
 - select cursor 2-244
 - sequential cursor 2-246
 - syntax 2-241
 - update cursor 2-245, 2-249
 - updating specified columns 2-250
 - use with concatenation operator 4-36
 - WHERE CURRENT OF clause 2-249
 - with SELECT 2-465
- DECODE function 4-45
- DEFAULT keyword
 - in CONNECT statement 2-79
 - in CREATE EXTERNAL TABLE 2-101
 - in CREATE PROCEDURE statement 2-137
 - in DISCONNECT statement 2-268
 - in SET CONNECTION statement 2-511
- Default locale Intro-4
- Default value
 - in ALTER TABLE 2-43
 - specifying with CREATE TABLE 2-160
- Deferred-Prepare feature
 - in SET DEFERRED_PREPARE 2-541
- DEFINE statement
 - default value clause 3-12
 - permissible data types 3-9
 - syntax 3-8
- DELETE keyword
 - in GRANT 2-353
 - in REVOKE 2-436
 - in REVOKE FRAGMENT 2-446
- DELETE statement
 - and triggers 2-200, 2-212
 - cascading 2-259
 - CURRENT OF clause 2-261
 - syntax 2-258
 - with cursor 2-249
 - with SELECT...FOR UPDATE 2-492
 - within a transaction 2-259
- DELIMIDENT environment
 - variable 4-118
- Delimited identifiers
 - multibyte characters in 4-118
 - non-ASCII characters in 4-118
 - syntax 4-116
- Delimiter
 - for LOAD input file 2-389
 - specifying with UNLOAD 2-619
- DELIMITER keyword
 - in CREATE EXTERNAL TABLE 2-101, 2-500
 - in LOAD 2-389
 - in UNLOAD 2-619
- DELUXE keyword, in CREATE EXTERNAL TABLE 2-101
- Demonstration database Intro-4
 - See also* stores7 database.
- DESC keyword
 - in CREATE INDEX 2-111
 - in SELECT 2-486, 2-489
 - order with nulls 2-490
- DESCRIBE statement
 - and the USING SQL DESCRIPTOR clause 2-265
 - describing statement type 2-263
- INTO sqllda pointer clause 2-266
- relation to GET DESCRIPTOR 2-322
- syntax 2-262
- using with concatenation operator 4-36
- values returned by SELECT 2-264
- Descriptor 2-264
- DETACH clause
 - hybrid fragmentation 2-20
 - in ALTER FRAGMENT 2-19
- Detached index 2-118
- Diagnostics table
 - creating with START VIOLATIONS TABLE 2-595
 - examples 2-527, 2-611, 2-613, 2-615
 - how to stop 2-614
 - privileges on 2-608
 - relationship to target table 2-600
 - relationship to violations table 2-600
 - restriction on dropping 2-281
 - structure 2-607
 - use with SET Database Object Mode statement 2-518
 - when to start 2-518
- Dirty Read isolation level 2-557
- DIRTY READ keywords, syntax in SET ISOLATION 2-556
- Disabled database object mode
 - benefits of 2-532
 - defined 2-520
- DISCONNECT statement
 - ALL keyword 2-270
 - CURRENT keyword 2-269
 - DEFAULT option 2-268
- Display labels
 - in CREATE VIEW 2-231
 - in SELECT 2-453, 2-461
- DISTINCT keyword
 - in aggregate expression 4-98
 - in CREATE INDEX 2-108
 - in SELECT 2-453, 2-457
 - in subquery 4-16
- Distributions
 - dropping with DROP DISTRIBUTIONS clause 2-640

- privileges required to create 2-641
- using the MEDIUM keyword 2-643
- Division (/) symbol, arithmetic operator 4-34
- DML statements, summary 1-10
- Documentation
 - on-line manuals Intro-16
- Documentation conventions
 - icon Intro-9
 - sample-code Intro-15
 - syntax Intro-11
 - typographical Intro-8
- Documentation notes Intro-18
- Documentation notes, program item Intro-19
- Documentation, types of
 - documentation notes Intro-18
 - error message files Intro-17
 - machine notes Intro-18
 - printed manuals Intro-17
 - related reading Intro-19
 - release notes Intro-18
- Double-dash (--) comment
 - symbol 1-6
- DROP clause, in ALTER TABLE 2-54
- DROP CONSTRAINT clause, in ALTER TABLE 2-63
- DROP CRCOLS clause, in ALTER TABLE 2-40
- DROP DATABASE
 - statement 2-271
- DROP INDEX statement,
 - syntax 2-273
- DROP ROLE statement 2-276
- DROP ROWIDS clause, in ALTER TABLE 2-40
- DROP SYNONYM statement,
 - syntax 2-277
- DROP TABLE statement,
 - syntax 2-279
- DROP TRIGGER statement
 - syntax 2-282
 - use of 2-282
- DROP VIEW statement,
 - syntax 2-283

- Duplicate values, in a query 2-457
- Dynamic management statements 1-10

E

- Enabled database object mode
 - benefits of 2-533
 - defined 2-520
- END keyword, in CREATE PROCEDURE 2-143
- Environment variable, setting with SYSTEM statement 3-45
- en_us.8859-1 locale Intro-4
- Error checking
 - continuing after error in stored procedure 3-38
 - error status with ON EXCEPTION 3-35
 - with SYSTEM 3-45
 - with WHENEVER 2-649
- Error message files Intro-17
- ESCAPE clause
 - in Condition segment 4-6, 4-12
 - with LIKE keyword 4-14
 - with MATCHES keyword 4-14
- ESCAPE keyword
 - in CREATE EXTERNAL TABLE 2-102, 2-500
 - with LIKE keyword 2-476
 - with MATCHES keyword 2-476
- EXCLUSIVE keyword
 - in DATABASE 2-236, 2-237
 - in LOCK TABLE 2-391, 2-392
- EXECUTE IMMEDIATE statement
 - restricted statement types 2-295
 - syntax and usage 2-294
 - using with concatenation operator 4-36
- EXECUTE ON keywords, in GRANT 2-343, 2-349
- EXECUTE PROCEDURE
 - statement 2-297
 - cursor, with DECLARE 2-244
 - in FOREACH 3-23
 - in triggered action 2-212
 - with INTO keyword 2-306

- EXECUTE statement and sqlca
 - record 2-287
- error conditions with 2-293
- INTO clause 2-287
- parameterizing a statement 2-290
- syntax 2-285
- USING DESCRIPTOR
 - clause 2-292
 - with concatenation operator 4-36
 - with USING keyword 2-290
- EXISTS keyword
 - beginning a subquery 2-477
 - in condition subquery 4-17
- EXIT statement 3-16
- EXP function 4-69
- Exponential function 4-69
- EXPRESS keyword, in CREATE EXTERNAL TABLE 2-102
- Expression
 - in UPDATE 2-630
 - ordering by 2-490
- Expression segment
 - aggregate expressions 4-98
 - column expressions 4-36
 - combined expressions 4-111
 - constant expressions 4-48
 - function expressions 4-55
 - in SPL expressions 4-112
 - syntax 4-34
- Expression-based distribution
 - scheme, in CREATE INDEX 2-125
- EXTEND function 4-73, 4-77
- Extension, to SQL
 - symbol for Intro-11, Intro-13
- Extent
 - modifying with ALTER TABLE 2-64
 - revising the size 2-65
- EXTENT SIZE keywords, in CREATE TABLE 2-189
- External tables
 - creating 2-92
 - integer data types 2-94
 - null values 2-95
 - use with SELECT 2-498

F

Feature icons Intro-10

Features, product Intro-5

FETCH statement

as affected by CLOSE 2-71

checking results with

SQLCA 2-310

fetching a row for update 2-309

locking for update 2-309

relation to GET

DESCRIPTOR 2-318

specifying memory location of a value 2-306

syntax 2-301

with

concatenation operator 4-36

program arrays 2-307

scroll cursor 2-304

sequential cursor 2-304

X/Open mode 2-303

Field qualifier

for DATETIME 4-32

for INTERVAL 4-131, 4-137

File

sending output with the OUTPUT statement 2-403

FILLFACTOR clause, in CREATE

INDEX 2-123

Filtering database object mode

benefits of 2-533

defined 2-520

finderr utility Intro-17

FIRST clause, in SELECT 2-455

FIRST keyword, in FETCH 2-304

FLOAT data type

syntax 4-27

using as default value 2-161

FLUSH statement

syntax 2-312

with concatenation operator 4-36

FOR EACH ROW clause

in CREATE TRIGGER 2-204, 2-205, 2-206

FOR keyword

in CONTINUE 3-7

in CREATE SYNONYM 2-151

in EXIT 3-16

FOR READ ONLY keywords

in DECLARE 2-251

FOR statement

specifying multiple ranges 3-21

syntax 3-18

using expression lists 3-21

using increments 3-20

FOR TABLE keywords, in UPDATE

STATISTICS 2-635

FOR UPDATE keywords

in DECLARE 2-241, 2-245, 2-249, 2-254

in SELECT 2-451, 2-491

relation to UPDATE 2-633

with column list 2-250

FOREACH keyword

in CONTINUE statement 3-7

in EXIT 3-16

FOREACH statement, syntax 3-23

Foreign key 2-166, 2-177

FORMAT keyword, in CREATE

EXTERNAL TABLE 2-102, 2-500

FRACTION keyword

as DATETIME field

qualifier 4-134

as INTERVAL field

qualifier 4-137

in DATETIME data type 4-32

in INTERVAL data type 4-131

FRAGMENT BY EXPRESSION

clause

in CREATE INDEX 2-125

Fragmentation

adding a fragment 2-28

adding rowids 2-40

adding rowids with CREATE

TABLE 2-179

altering fragments 2-8

arbitrary rule 2-184

attaching tables 2-11

combining tables 2-11

defining a new strategy 2-11

detaching a table fragment 2-19

dropping an existing

fragment 2-30

dropping rowids 2-40

fragment expressions 2-17

if you run out of log / disk

space 2-10

modifying an existing fragment

expression 2-31

modifying indexes 2-33

number of rows in fragment 2-11

of indexes 2-125

of tables 2-181

of temporary tables 2-191

reinitializing strategy 2-26

remainder 2-30

reverting to non-fragmented 2-25

rowid 2-26

skipping an unavailable

dbspace 2-535

strategy

by expression 2-184

range rule 2-184

round robin 2-183

system-defined hash distribution

scheme 2-184

text and byte data types 2-18

Fragment-level privilege

granting with GRANT

FRAGMENT 2-360

revoking 2-444, 2-446

FREE statement

effect on cursors 2-401

syntax 2-315

with concatenation operator 4-36

FROM clause, in SELECT 2-451,

2-467

FROM keyword, in PUT 2-422

Function expression

Algebraic functions 4-55, 4-56

CHARACTER_LENGTH 4-72

DBINFO function 4-61

description of 4-55

Exponential and logarithmic

functions 4-55

HEX function 4-55

in SELECT 2-459

LENGTH function 4-55, 4-71

OCTET_LENGTH 4-72

Time functions 4-55

Trigonometric functions 4-55

TRIM function 4-55

G

Generalized-key index
 description of 2-119
 SELECT clause 2-120
 WHERE clause 2-122

GET DESCRIPTOR statement
 syntax 2-318
 the COUNT keyword 2-321
 use with FETCH statement 2-308
 with concatenation operator 4-36

GET DIAGNOSTICS statement
 CLASS_ORIGIN keyword 2-335
 CONNECTION_NAME
 keyword 2-338
 exception clause 2-333
 MESSAGE_LENGTH
 keyword 2-335
 MESSAGE_TEXT keyword 2-335
 MORE keyword 2-332
 NUMBER keyword 2-332
 purpose 2-325
 RETURNED_SQLSTATE
 keyword 2-335
 ROW_COUNT keyword 2-332
 SERVER_NAME keyword 2-335
 SQLSTATE codes 2-327
 statement clause 2-331
 SUBCLASS_ORIGIN
 keyword 2-335
 syntax 2-325
 with concatenation operator 4-36

GK index. *See* Generalized-key index.

Global environment,
 definition 3-10

Global Language Support
 (GLS) Intro-4

GLS. *See* Global Language Support.

GL_DATE environment
 variable 2-386, 2-617

GL_DATETIME environment
 variable 2-387, 2-618

GO TO keywords, in the
 WHENEVER statement 2-646,
 2-651

GRANT FRAGMENT statement

 AS grantor clause 2-367
 syntax 2-360
 WITH GRANT OPTION
 clause 2-366

GRANT statement
 ALL keyword 2-353, 2-356
 changing grantor 2-358
 database-level privileges 2-346
 default-table privileges 2-356
 DELETE keyword 2-353
 INDEX keyword 2-353
 INSERT keyword 2-353
 passing grant ability 2-357
 PRIVILEGES keyword 2-353
 privileges on a view 2-359
 REFERENCES keyword 2-353
 role privileges 2-349
 SELECT keyword 2-353
 syntax 2-344
 table-level privileges 2-354
 UPDATE keyword 2-353
 WITH GRANT OPTION
 keywords 2-357

GROUP BY clause, in
 SELECT 2-451, 2-482

H

HAVING clause, in SELECT 2-451,
 2-484

HEX function 4-39, 4-70

HIGH keyword, in SET
 OPTIMIZATION 2-567

Hold cursor
 definition of 2-246
 insert cursor with hold 2-257
 use of 2-247

HOURL keyword
 in DATETIME 4-32
 in DATETIME segment 4-134
 in INTERVAL 4-137
 in INTERVAL data type 4-131

HYBRID clause, in CREATE
 TABLE 2-187

I

IBM-format binary, in external
 tables 2-94

Icons
 comment Intro-9
 compliance Intro-11
 feature Intro-10
 platform Intro-10
 product Intro-10
 syntax diagram Intro-13

IDATA field
 SET DESCRIPTOR
 statement 2-550
 with X/Open programs 2-322

Identifier segment
 column names 4-124
 cursor name 4-126, 4-129
 delimited identifiers 4-116
 multibyte characters in 4-116
 non-ASCII characters in 4-116
 stored procedures 4-126
 syntax 4-114
 table names 4-123, 4-125
 uppercase characters in 4-115
 using keywords as column
 names 4-121
 variable name 4-128

IF statement
 syntax 3-27
 with null values 3-28

ILENGTH field
 SET DESCRIPTOR
 statement 2-550
 with X/Open programs 2-322

IN keyword
 in Condition segment 4-11
 in Condition subquery 4-16
 in LOCK TABLE 2-391
 in WHERE clause 2-474

Index
 attached, defined 2-118
 bidirectional traversal 2-111
 cleaner list. *See* B-tree cleaner list.
 clustered fragments 2-109
 composite 2-110
 converting during
 upgrade 2-635, 2-639
 creating fragments 2-125

- creating with CREATE
 - INDEX 2-106
- detached 2-118
- disabled 2-132
- displaying information for 2-371
- Dropping with DROP
 - INDEX 2-273
- duplicate 2-108
- effects of ALTER FRAGMENT
 - ATTACH 2-13
- effects of altered fragments 2-18
- name, specifying 4-25
- number allowed on same
 - columns 2-116
- on fragmented tables 2-109
- on ORDER BY columns 2-112, 2-491
- provide for expansion 2-123
- resident in memory 2-576
- sharing with constraints 2-158
- unique 2-131
 - adding when duplicate values exist 2-525
 - restrictions 2-27
- with temporary tables 2-497
- with unique and referential constraints 2-109
- INDEX keyword
 - in GRANT 2-353
 - in REVOKE 2-436
- Index Name. *See* Database Object Name.
- Index privilege 2-355
- INDEXES FOR keywords, in INFO statement 2-371
- INDICATOR field, setting with SET DESCRIPTOR 2-552
- INDICATOR keyword, in SELECT 2-464, 2-465, 2-466
- Indicator variable
 - in expression 4-111
 - in SELECT 2-464, 2-465, 2-466
- Industry standards, compliance with Intro-20
- INFO statement, syntax 2-369
- INFORMIXDIR/bin
 - directory Intro-5
- INFORMIXSERVER environment variable 2-79

- informix, privileges associated with user 2-348
- Insert buffer
 - counting inserted rows 2-314, 2-425
 - filling with constant values 2-420
 - inserting rows with a cursor 2-376
 - storing rows with PUT 2-419
 - triggering flushing 2-423
- Insert cursor
 - closing 2-71
 - in INSERT 2-376
 - in PUT 2-421
 - opening 2-397
 - reopening 2-398
 - result of CLOSE in SQLCA 2-71
 - use of 2-245
 - with hold 2-257
- INSERT INTO keywords
 - in INSERT 2-373
 - in LOAD 2-390
- INSERT keyword
 - in GRANT 2-353
 - in REVOKE 2-436
 - in REVOKE FRAGMENT 2-446
- INSERT statement
 - and triggers 2-200, 2-212
 - effect of transactions 2-377
 - filling insert buffer with PUT 2-419
 - in dynamic SQL 2-382
 - inserting
 - rows through a view 2-375
 - rows with a cursor 2-376
 - inserting nulls 2-381
 - SERIAL columns 2-380
 - specifying values to insert 2-378
 - syntax 2-373
 - using functions 2-380
 - with DECLARE 2-241
 - with insert cursor 2-256
 - with SELECT 2-381
- INTEGER data type
 - syntax 4-27
 - using as default value 2-161
- Integers, in external tables 2-94
- Integrity. *See* Data integrity.

- INTERVAL data type
 - as quoted string 4-160
 - field qualifier, syntax 4-131
 - in expression 4-54
 - in INSERT 4-161
 - syntax 4-27, 4-137
 - using as default value 2-161
- INTO clause, in SELECT 2-462
- INTO EXTERNAL clause, in SELECT 2-498
- INTO keyword, in FETCH 2-307
- INTO SCRATCH clause, in SELECT 2-498
- INTO TEMP clause
 - in SELECT 2-451, 2-497
 - with UNION operator 2-501
- IS keyword, in WHERE clause 2-475
- IS NOT keywords, syntax in Condition segment 4-6
- IS NOT NULL keywords, in Condition segment 4-11
- IS NULL keywords, in Condition segment 4-11
- ISAM error code 3-34, 3-39
- ISO 8859-1 code set Intro-4
- Isolating a table from other tables 2-179, 2-183
- Isolation level
 - Committed Read 2-557
 - Cursor Stability 2-558
 - definitions 2-557, 2-588
 - Dirty Read 2-557
 - in external tables 2-559, 2-585
 - Read Committed 2-588
 - Read Uncommitted 2-588
 - Repeatable Read 2-558, 2-588
 - Serializable 2-588
 - use with FETCH statement 2-309
- Item descriptor, defined 2-7
- ITYPE field
 - SET DESCRIPTOR statement 2-550
 - setting with SET DESCRIPTOR 2-552
 - with X/Open programs 2-322

J

Join

- in Condition segment 2-479
- multiple-table join 2-480
- outer 2-468, 2-481
- self-join 2-481
- two-table join 2-480

Join column. *See* Foreign key.

Join on key query 2-121

K

Keywords

- SQL 4-115, A-1
- using in triggered action 2-213

L

LAST keyword, in FETCH 2-304

LENGTH field

- setting with SET DESCRIPTOR 2-551
- with DATETIME and INTERVAL types 2-552
- with DECIMAL and MONEY types 2-551

LENGTH function 2-459, 4-71, 4-72

LET statement, syntax 3-31

LIKE keyword

- in Condition segment 4-6
- in SELECT 2-475
- wildcard characters 2-476

List-mode format, in SET Database Object Mode statement 2-515

Literal

DATETIME

- in Condition segment 4-6
- in expression 4-48, 4-53
- in INSERT 2-378
- segment 4-134
- with IN keyword 2-474

DATETIME in ALTER

TABLE 2-43

INTERVAL

- in Condition segment 4-6
- in expression 4-48, 4-54

in INSERT 2-378

segment 4-137

Number

- in Condition segment 4-6
- in expression 4-48, 4-52
- in INSERT 2-378
- segment 4-140
- with IN keyword 4-11
- using as default value 2-161

LOAD statement

DELIMITER clause 2-389

INSERT INTO clause 2-390

specifying the table to load into 2-390

syntax 2-384

the LOAD FROM file 2-386

VARCHAR, TEXT, or BYTE data types 2-388

Locale Intro-4

LOCK MODE clause

- in ALTER TABLE 2-65
- in CREATE TABLE 2-188

LOCK TABLE statement

- in databases with transactions 2-392
- in databases without transactions 2-393
- syntax 2-391
- use in transactions 2-67

Locking

default table locking 2-188

during

- inserts 2-377
- updates 2-249, 2-627

exclusive locks 2-391

in transaction 2-67

overriding row-level 2-392

releasing with COMMIT

WORK 2-75

releasing with ROLLBACK

WORK 2-449

shared locks 2-391

types of locks 2-65, 2-188

update cursors effect on 2-249

update locks 2-627

waiting period 2-562

with

FETCH 2-309

SET ISOLATION 2-556

SET LOCK MODE 2-561

SET TRANSACTION 2-585

UNLOCK TABLE 2-621

Log file, for load and unload jobs 2-574

LOG10 function 4-69, 4-70

Logarithmic functions

LOG10 function 4-69, 4-70

LOGN function 4-69, 4-70

Logging

buffered versus unbuffered 2-564

cascading deletes 2-51, 2-170

changing mode with SET

LOG 2-564

in CREATE DATABASE

statement 2-90

with triggers 2-227

Logical operator, in Condition segment 4-20

LOGN function 4-69, 4-70

Loop

controlled 3-18

indefinite with WHILE 3-50

LOW keyword, in SET

OPTIMIZATION 2-567

LPAD function 4-93

M

Machine notes Intro-18

Mail, sending from stored procedure 3-45

MATCHES keyword

in Condition segment 4-6, 4-12

in SELECT 2-475

wildcard characters 2-476

MAX function 4-98, 4-105

MAXERRORS environment variable 2-101

MAXERRORS keyword

in CREATE EXTERNAL TABLE 2-102

MDY function 4-73, 4-78

Message file

error messages Intro-17

MIN function 4-98, 4-106

Minus (-) sign, arithmetic operator 4-34

MINUTE keyword
 in DATETIME data type 4-32
 in DATETIME segment 4-134
 in INTERVAL 4-131, 4-137

MOD function 4-56, 4-59

MODE ANSI keywords
 See also ANSI.

Model. *See* Data model.

MODIFY clause, in ALTER TABLE 2-56

MODIFY NEXT SIZE clause, in ALTER TABLE 2-64

MONEY data type
 syntax 4-27
 using as default value 2-161

MONTH function 4-73, 4-76

MONTH keyword
 in DATETIME data type 4-32
 in DATETIME segment 4-134
 in INTERVAL 4-131, 4-137

MS-DOS operating system. *See* DOS operating system.

Multiple triggers
 column numbers in 2-203
 example 2-203
 order of execution 2-203
 preventing overriding 2-226

Multiplication sign (*), arithmetic operator 4-34

Multirow query, destination of returned values 2-306

N

Naming convention
 database 4-23
 database objects 4-25
 table 2-156, 4-25

Nested ordering, in SELECT 2-490

NEW keyword, in REFERENCING clauses 2-207, 2-209, 2-210

NEXT keyword, in FETCH 2-304

NEXT SIZE keywords, in CREATE TABLE 2-189

NODEFDAC environment variable, effects on new stored procedure 2-135

Noncursory procedure 2-138

NOT CLUSTER keywords, in ALTER INDEX 2-35

NOT FOUND keywords in WHENEVER statement 2-646

NOT IN keywords 4-16

NOT keyword
 in Condition segment 4-5, 4-6, 4-12
 in WHERE clause 2-475
 with BETWEEN keyword 2-474
 with IN keyword 2-477

NOT NULL keywords
 in ALTER TABLE 2-44
 in CREATE EXTERNAL TABLE 2-96
 in CREATE TABLE 2-164

NOT WAIT keywords in SET LOCK MODE 2-562

NULL keyword, ambiguous as procedure variable 4-127

Null value
 checking for in SELECT 2-288, 2-291
 in SPL IF statement 3-28
 inserting with the VALUES clause 2-381
 returned implicitly by stored procedure 3-41
 updating a column 2-630
 used in Condition with NOT operator 4-6
 used in the ORDER BY clause 2-490
 with AND and OR keywords 4-6
 with SPL WHILE statement 3-50

NVARCHAR data type, syntax 4-29

NVL function 4-44

O

Object mode. *See* Database object mode.

OCTET_LENGTH function 4-72

OF keyword
 in DECLARE 2-250
 syntax in DECLARE 2-241

OLD keyword, in REFERENCING clause 2-207, 2-209, 2-210

ON DELETE CASCADE keywords 2-50, 2-200

ON EXCEPTION statement
 placement of 3-35
 syntax 3-34

ON keyword
 in CREATE INDEX 2-106
 in GRANT 2-344

On-line manuals Intro-16

OPEN statement
 constructing the active set 2-395
 opening a procedure cursor 2-396
 opening an insert cursor 2-397
 opening select or update cursors 2-395
 reopening a cursor 2-398
 substituting values for ? parameters 2-398
 syntax 2-394
 with concatenation operator 4-36
 with FREE 2-401

OPERATIONAL table type, creation of 2-178

Optical Subsystem
 list of statements 1-11

Optimization, specifying a high or low level 2-566

Optimizer
 and Optimizer Directives segment 4-142
 and SET OPTIMIZATION statement 2-566
 with UPDATE STATISTICS 2-637

Optimizer Directives
 segment 4-142

Optimizing
 a query 2-553
 a server 2-566
 across a network 2-568

OR keyword 4-5, 4-20

ORDER BY clause
 in SELECT 2-451, 2-486, 2-489
 indexes on columns 2-491
 select columns by number 2-490
 with UNION operator 2-501

- ORDER BY keywords
 - indexes on ORDER BY
 - columns 2-112
 - restrictions in INSERT 2-382
- Order of execution, of action
 - statements 2-212
- Outer join. *See* Join, outer.
- OUTPUT statement, syntax and
 - use 2-402
- Owner
 - case-sensitivity 4-156
 - in ALTER TABLE 2-38, 2-62
 - in ANSI-compliant
 - database 4-155
 - in CREATE SYNONYM 2-152
 - in Database Object Name
 - segment 4-155
 - in Object Name segment 4-25,
 - 4-142, 4-146, 4-150, 4-153
 - in Owner Name segment 4-155
 - in RENAME COLUMN 2-426
 - in RENAME TABLE 2-429
 - in system catalog table 2-63
- Owner Name segment 4-155
- Owner-privileged procedure 2-135

P

- Packed decimal, in external
 - tables 2-94
- PAGE keyword, in CREATE
 - TABLE 2-65, 2-188
- Parallel distributed queries, SET
 - PRIORITY statement 2-570
- Parameter
 - BYTE or TEXT in SPL 3-15
 - in CALL statement 3-5
 - to a stored procedure 2-136
- Parameterizing
 - defined 2-290
 - prepared statements 2-290
- Parameterizing a statement, with
 - SQL identifiers 2-411
- Parent-child relationship 2-49,
 - 2-166
- PDQ, SET PDQPRIORITY
 - statement 2-570
- Percent (%) sign, as wildcard 4-12

- Permission. *See also* Privilege.
- Phantom row 2-557, 2-588
- Pipe character 2-100, 2-499
- PIPE keyword, in the OUTPUT
 - statement 2-403
- Platform icons Intro-10
- Plus (+) sign, arithmetic
 - operator 4-34
- POW function 4-56, 4-59
- PRECISION field
 - with GET DESCRIPTOR 2-323
 - with SET DESCRIPTOR 2-551
- PREPARE statement
 - deferring 2-541
 - increasing efficiency 2-416
 - multistatement text 2-295, 2-414,
 - 2-416
 - parameterizing a statement 2-410
 - parameterizing for SQL
 - identifiers 2-411
 - question (?) mark as
 - placeholder 2-405
 - releasing resources with
 - FREE 2-316
 - restrictions with SELECT 2-407
 - statement identifier use 2-405
 - syntax 2-404
 - valid statement text 2-407
 - with concatenation operator 4-36
- Prepared statement
 - comment symbols in 2-407
 - describing returned values with
 - DESCRIBE 2-263
 - executing 2-285
 - parameterizing 2-290
 - prepared object limit 2-405
 - valid statement text 2-407
- PREVIOUS keyword, in
 - FETCH 2-304
- Primary key constraint 2-49
 - data type conversion 2-58
- defining column as 2-176
- dropping 2-64
- enforcing 2-158
- referencing 2-49
- requirements for 2-44, 2-176
- rules of use 2-166
- Printed manuals Intro-17

- PRIOR keyword, in FETCH 2-304
- Privilege
 - Connect 2-347
 - revoking with REVOKE 2-433
 - when privileges conflict 2-345
- Privileges
 - DBA 2-348
 - default for table using CREATE
 - TABLE 2-156
 - defined with GRANT
 - statement 2-353
 - for triggered action 2-222
 - fragment-level
 - defined 2-362
 - duration of 2-364
 - granting with GRANT
 - FRAGMENT 2-360
 - revoking 2-444
 - role in command
 - validation 2-363
 - granting to roles 2-349
 - granting with GRANT 2-343
 - needed
 - to create a view 2-359
 - to drop an index 2-273
 - on a synonym 2-151
 - on a table fragment 2-360
 - on a view 2-231
 - Resource 2-347
 - revoking column-specific 2-439
 - revoking with REVOKE 2-432
- PRIVILEGES FOR keywords, in
 - INFO statement 2-371
- PRIVILEGES keyword
 - in GRANT 2-353
 - in REVOKE 2-436
- Procedure cursor
 - opening 2-396
 - reopening 2-398
- Procedure name, specifying 4-25
- Procedure Name. *See* Database
 - Object Name.
- Procedure, stored. *See* Stored
 - procedure.
- Product icons Intro-10
- Program group
 - Documentation notes Intro-19
- Promotable lock 2-249

PUBLIC keyword
 in GRANT 2-344, 2-348
 in REVOKE 2-437
PUT statement
 effect on trigger 2-201
 source of row values 2-420
 syntax 2-418
 use in transactions 2-419
 with concatenation operator 4-36

Q

Qualifier, field
 for DATETIME 4-32, 4-134
 for INTERVAL 4-131, 4-137
Query
 optimizing with Optimizer
 Directives 4-142
 optimizing with SET
 OPTIMIZATION 2-566
 piping results to another
 program 2-403
 scheduling level for 2-581
 sending results to an operating
 system file 2-403
 sending results to another
 program 2-403
Question mark (?)
 as placeholder in PREPARE 2-405
 as wildcard 4-13
 naming variables in PUT 2-422
 replacing with USING
 keyword 2-398
Quotation marks 4-116
 double
 delimited identifier 4-119
 in owner name 4-156
 in quoted string 4-158, 4-159
Quoted string
 DATETIME, INTERVAL values
 as strings 4-160
 in Condition segment 4-6
 in expression 4-48, 4-50
 in INSERT 2-378, 4-161
 syntax 4-158
 wildcards 4-160
 with LIKE keywords 2-475

R

RAISE EXCEPTION statement,
 syntax 3-39
RANGE function 4-106
Range rule 2-127
RAW table type, creation of 2-177
Read Committed isolation
 level 2-588
READ COMMITTED keywords,
 SET TRANSACTION 2-585
Read Uncommitted isolation
 level 2-588
READ UNCOMMITTED
 keywords, SET
 TRANSACTION 2-585
RECORDEND environment
 variable 2-101, 2-499
RECORDEND keyword in
 CREATE EXTERNAL
 TABLE 2-102, 2-500
REFERENCES clause
 in ALTER TABLE 2-48
 in CREATE TABLE 2-164
REFERENCES keyword
 in GRANT 2-353
 in REVOKE 2-436
References privilege, definition
 of 2-355
REFERENCING clause
 DELETE REFERENCING
 clause 2-208
 INSERT REFERENCING
 clause 2-207
 UPDATE REFERENCING
 clause 2-209
 using referencing 2-214
REFERENCING clause, in CREATE
 TRIGGER 2-210
Referential constraint
 and a DELETE trigger 2-200
 data type restrictions 2-167
 definition of 2-49
 dropping 2-64
 enforcing 2-158
 rules of use 2-166
REJECTFILE keyword, in CREATE
 EXTERNAL TABLE 2-100,
 2-102, 2-103

Related reading Intro-19
Relational operator
 in Condition segment 4-6
 segment 4-162
 with WHERE keyword in
 SELECT 2-473
RELATIVE keyword, in
 FETCH 2-305
Release notes Intro-18, Intro-19
Remainder, in fragment
 expressions 2-30
RENAME COLUMN statement
 restrictions 2-426
 syntax 2-426
RENAME DATABASE statement
 syntax 2-428
RENAME TABLE statement
 ANSI-compliant naming 2-429
 syntax 2-429
Repeatable Read isolation level
 description of 2-558, 2-588
 emulating during update 2-310
REPEATABLE READ keywords
 syntax in SET ISOLATION 2-556
 syntax in SET
 TRANSACTION 2-585
REPLACE function 4-91
Reserved words
 listed A-1
 with delimited identifiers 4-117
 with identifiers 4-115
Resolution
 in UPDATE STATISTICS 2-641,
 2-644
 with data distributions 2-641
RESOURCE keyword, in
 REVOKE 2-440
RESOURCE permission, with
 CREATE SCHEMA 2-149
Resource privilege 2-347
RESTRICT keyword, in
 REVOKE 2-434, 2-435
RETURN statement
 returning insufficient values 3-41
 returning null values 3-41
 syntax 3-41
REVOKE FRAGMENT statement
 ALL keyword 2-446
 DELETE keyword 2-446

- INSERT keyword 2-446
- revoking privileges 2-446
- syntax 2-444
- UPDATE keyword 2-446
- REVOKE statement
 - ALL keyword 2-436
 - ALTER keyword 2-436
 - column-specific privileges 2-439
 - database-level privileges 2-440
 - DBA keyword 2-440
 - DELETE keyword 2-436
 - in system catalog tables 2-437
 - INDEX keyword 2-436
 - INSERT keyword 2-436
 - PRIVILEGES keyword 2-436
 - privileges needed 2-433
 - PUBLIC keyword 2-437
 - REFERENCES keyword 2-436
 - RESOURCE keyword 2-440
 - RESTRICT keyword 2-435
 - SELECT keyword 2-436
 - syntax 2-432
 - table-level privileges 2-436
 - UPDATE keyword 2-436
 - using with roles 2-434
- Role
 - creating with CREATE ROLE statement 2-146
 - definition 2-146
 - dropping with DROP ROLE statement 2-276
 - enabling with SET ROLE 2-579
 - granting privileges with GRANT statement 2-349
 - revoking privileges with REVOKE 2-434
 - scope of 2-580
 - setting with SET ROLE 2-579
- ROLLBACK WORK statement
 - syntax 2-449
 - with WHENEVER 2-67, 2-74, 2-450
- ROOT function 4-56, 4-59
- ROUND function 4-56, 4-59
- Row
 - deleting 2-258
 - engine response to locked row 2-562
 - finding location of 4-39

- inserting
 - through a view 2-375
 - with a cursor 2-376
- order, guaranteeing
 - independence of 2-206
- phantom 2-557, 2-588
- retrieving with FETCH 2-305
- rowid definition 2-305
- updating through a view 2-625
- writing buffered rows with FLUSH 2-312
- ROW keyword, in CREATE TABLE 2-65, 2-188
- Rowid
 - adding column with INIT clause 2-26
 - adding with ALTER TABLE 2-40
 - dropping from fragmented tables 2-40
 - use in a column expression 4-39
 - use in fragmented tables 2-26
 - used as column name 4-123
- ROWID keyword 4-39
- RPAD function 4-94
- Rule
 - arbitrary 2-127
 - range 2-127
- Rules for stored procedures 2-222

S

- SAMEAS clause 2-94
- Sample-code conventions Intro-15
- SCALE field
 - with GET DESCRIPTOR 2-323
 - with SET DESCRIPTOR 2-551
- Schema. *See* Data model.
- Scratch tables 2-191, 2-498
- Scroll cursor
 - definition of 2-246
 - FETCH with 2-304
 - use of 2-246
- SCROLL keyword
 - in DECLARE 2-246
 - syntax in DECLARE 2-241
- SECOND keyword
 - as DATETIME field
 - qualifier 4-134
- as INTERVAL field
 - qualifier 4-137
- in DATETIME data type 4-32
- in INTERVAL data type 4-131
- Segment
 - defined 4-3
 - relation to SPL statements 4-3
 - relation to SQL statements 4-3
- SELECT
 - INTO EXTERNAL clause 2-498
 - INTO SCRATCH clause 2-498
 - WITH NO LOG keywords 2-498
- Select cursor
 - opening 2-395
 - reopening 2-398
 - use of 2-244
- SELECT keyword
 - ambiguous use as procedure variable 4-127
 - in GRANT 2-353
 - in REVOKE 2-436
- SELECT statement
 - aggregate functions in 4-98
 - as an argument to a stored procedure 3-6
 - BETWEEN condition 2-474
 - column numbers 2-490
 - cursor for 2-491, 2-493
 - cursor, with DECLARE 2-244
 - describing returned values with DESCRIBE 2-262, 2-264, 2-265, 2-266
 - FIRST clause 2-455
 - FOR READ ONLY clause 2-493
 - FOR UPDATE clause 2-491
 - FROM Clause 2-467
 - GROUP BY clause 2-482
 - HAVING clause 2-484
 - IN condition 2-474
 - in FOR EACH ROW trigger 2-206
 - in INSERT 2-381
 - INTO clause with ESQL 2-462
 - INTO TEMP clause 2-497
 - IS NULL condition 2-475
 - joining tables in WHERE clause 2-479
 - LIKE or MATCHES condition 2-475

- null values in the ORDER BY clause 2-490
 - ORDER BY clause 2-486
 - outer join 2-468
 - relational-operator
 - condition 2-473
 - restrictions with INTO clause 2-407
 - ROWID keyword 4-39
 - SELECT clause 2-453
 - select numbers 2-490
 - singleton 2-463
 - stored procedure in 2-459
 - subquery with WHERE keyword 2-473
 - syntax 2-451
 - UNION operator 2-500
 - use of expressions 2-457
 - with
 - DECLARE 2-241
 - FOREACH 3-23
 - INTO keyword 2-306
 - LET 3-32
 - writing rows retrieved to an ASCII file 2-616
- Self-join, description of 2-481
- Sequential cursor
 - definition of 2-246
 - use of 2-246
 - with FETCH 2-304
- Serial column
 - nonsequential numbers in 2-186
 - use with hash fragmentation 2-186
- SERIAL data type
 - in INSERT 2-380
 - resetting values 2-58
 - syntax 4-27
- Serializable isolation level, description of 2-588
- SERIALIZABLE keyword, syntax in SET TRANSACTION 2-585
- Server. *See* Database server.
- Session control block
 - accessed by DBINFO function 4-65
 - defined 4-65
- Session id, returned by DBINFO function 4-65

- SET AUTOFREE statement
 - and associated prepared statement 2-504
 - and detached prepared statement 2-504
 - syntax and use 2-503
- SET clause 2-214
- SET CONNECTION statement
 - CURRENT keyword 2-511
 - DEFAULT option 2-511
 - syntax and use 2-506
 - with concatenation operator 4-36
- SET CONSTRAINT. *See* SET Database Object Mode statement and SET Transaction Mode statement.
- SET Database Object Mode statement
 - error options 2-517
 - list-mode format 2-515
 - relationship to START VIOLATIONS TABLE 2-518, 2-597
 - syntax 2-513
 - table-mode format 2-514
 - use
 - with data definition statements 2-525
 - with data manipulation statements 2-519
 - with diagnostics tables 2-518
 - with violations tables 2-518
 - with CREATE TRIGGER 2-225
- SET DATASKIP statement
 - syntax 2-535
 - what causes a skip 2-536
- SET DEBUG FILE TO statement
 - syntax and use 2-538
 - with TRACE 3-47
- SET DEFERRED_PREPARE statement
 - syntax 2-541
- SET DESCRIPTOR statement
 - syntax 2-548
 - the VALUE clause 2-549
 - with concatenation operator 4-36
 - X/Open mode 2-550
- SET EXPLAIN statement
 - interpreting output 2-554

- SET INDEX. *See* SET Database Object Mode statement and SET Residency statement.
- SET ISOLATION statement
 - default database levels 2-559
 - definition of isolation levels 2-557
 - effects of isolation 2-559
 - similarities to SET TRANSACTION statement 2-586
 - syntax 2-556
- SET keyword
 - syntax in UPDATE 2-623
 - use in UPDATE 2-628
- SET LOCK MODE statement
 - setting wait period 2-562
 - syntax 2-561
- SET LOG statement
 - buffered vs. unbuffered 2-564
 - syntax 2-564
- SET OPTIMIZATION statement
 - ALL_ROWS option 2-567
 - FIRST_ROWS option 2-567
 - HIGH option 2-567
 - LOW option 2-567
 - syntax and use 2-566
- SET PDQPRIORITY statement
 - syntax 2-570
- SET PLOAD FILE statement
 - syntax 2-574
- SET Residency statement
 - syntax 2-576
- SET ROLE statement 2-579
- SET SCHEDULE LEVEL statement
 - syntax 2-581
- SET SESSION AUTHORIZATION statement
 - scope 2-583
 - syntax 2-582
- SET statement. *See* SET Database Object Mode statement and SET Transaction Mode statement.
- SET TABLE. *See* SET Residency statement.
- SET Transaction Mode statement, syntax 2-591
- SET TRANSACTION statement
 - default database levels 2-589
 - definition of isolation levels 2-588

- effects of isolation 2-590
- similarities to SET ISOLATION statement 2-586
- syntax 2-585
- SET TRIGGER. *See* SET Database Object Mode statement.
- setnet32 2-83
- SHARE keyword, syntax in LOCK TABLE 2-391
- Simple assignment, in SPL 3-32
- SIN function 4-81
- Single-threaded application 2-509
- Singleton SELECT statement 2-463
- SIZE keyword, in CREATE EXTERNAL TABLE 2-102
- SMALLFLOAT data type, syntax 4-27
- SMALLINT data type syntax 4-27
 - using as default value 2-161
- Software dependencies Intro-4
- SOME keyword
 - beginning a subquery 2-478
 - in condition expression 4-18
- Sorting
 - in a combined query 2-501
 - in SELECT 2-486
- SPL statements, description 3-3
- SQL
 - comments 1-6
 - compliance of statements with ANSI standard 1-12
 - keywords 4-115, A-1
 - statement types 1-9
- SQL Communications Area (SQLCA)
 - and EXECUTE statement 2-287
 - result after CLOSE 2-71
 - result after DATABASE 2-237
 - result after DESCRIBE 2-263
 - result after FETCH 2-310
 - result after FLUSH 2-313
 - result after OPEN 2-397
 - result after PUT 2-424
 - result after SELECT 2-466
 - warning when dbspace skipped 2-535
- SQL statement. *See* Statement, SQL.
- SQLCA. *See* SQL Communications Area.
- sqlda structure
 - in DESCRIBE 2-263, 2-264
 - in EXECUTE 2-291
 - in FETCH 2-309
 - in OPEN 2-288, 2-291, 2-395, 2-400, 2-419
 - in PUT 2-422
- SQLERROR keyword, in the WHENEVER statement 2-646, 2-649
- sqlhosts file 2-83
- SQLNOTFOUND, error conditions with EXECUTE statement 2-293
- SQLSTATE
 - Not Found condition 2-650
 - runtime errors 2-649
 - warnings 2-649
- SQLSTATE variable
 - list of codes 2-327
- SQLWARNING keyword, in the WHENEVER statement 2-646, 2-649
- SQRT function 4-56, 4-60
- Square brackets [...]
 - as wildcard characters 4-13
- STANDARD table type, creation of 2-178
- START VIOLATIONS TABLE statement
 - privileges required for executing 2-599
 - relationship to SET Database Object Mode statement 2-518, 2-597
 - starting and stopping 2-597
 - syntax 2-595
- Statement identifier
 - associating with cursor 2-244
 - definition of 2-405
 - in DECLARE 2-241, 2-254
 - in FREE 2-315, 2-316
 - in PREPARE 2-405
 - releasing 2-406
- Statement, SQL
 - ANSI-compliant 1-12
 - extensions to ANSI standard 1-13
 - how to enter 1-3
 - types 1-9
- STATIC table type, creation of 2-177
- STATUS FOR keywords, in INFO statement 2-372
- Status, displaying with INFO statement 2-372
- STDEV function 4-107
- STOP keyword, in the WHENEVER statement 2-646, 2-650
- STOP VIOLATIONS TABLE statement
 - description of 2-614
 - privileges required for executing 2-615
 - syntax 2-614
- Storage options, in CREATE INDEX 2-122
- Stored procedure
 - as triggered action 2-222
- BEGIN-END block 2-143
- BYTE and TEXT data types 3-15
- checking references 2-222
- comments in 1-7, 2-139
- cursors with 3-24
- cursor 2-138, 3-25
- DBA-privileged, how to create 2-135
- DBA-privileged, use with triggers 2-223
- debugging 3-47
- definition of 3-3
- displaying documentation 2-139
- executing operating system commands from 3-44
- granting privileges on 2-349
- handling multiple rows 3-42
- header 3-9
- in SELECT statements 2-459
- in triggered action 2-212
- limits on return values 2-139
- naming output file for TRACE statement 2-538
- noncursory 2-138
- owner-privileged 2-135, 2-223
- privileges 2-223
- receiving data from SELECT 2-462
- revoking privileges on 2-434

- sending mail from 3-45
- simulating errors 3-39
- transaction in 2-143
 - See also* Procedure.
- stores7 database Intro-5
 - See also* Demonstration database.
- String-manipulation functions 4-84
- Structured Query Language. *See* SQL.
- Subquery
 - beginning with
 - ALL / ANY / SOME keywords 2-478
 - beginning with EXISTS
 - keyword 2-477
 - beginning with IN
 - keyword 2-477
 - correlated 4-15
 - definition of 2-473
 - in Condition segment 4-15
 - restrictions with UNION operator 2-501
 - with DISTINCT keyword 2-457
- Subscripting, on character columns 2-488, 4-38
- Substring
 - in column expression 4-38
 - in ORDER BY clause of SELECT 2-488
- SUBSTRING function 4-91, 4-93, 4-94
- SUM function 4-98, 4-106
- Surviving table 2-12
- Synonym
 - ANSI-compliant naming 2-152
 - chains 2-153
 - creating with CREATE SYNONYM 2-151
 - difference from alias 2-151
 - dropping 2-277
 - name, specifying 4-25
- Synonym Name. *See* Database Object Name.
- Syntax conventions
 - elements of Intro-12
 - example diagram Intro-14
 - how to read Intro-14
 - icons used in Intro-13

- syscolauth system catalog table 2-437
- sysdepend system catalog table 2-283
- systabauth system catalog table 2-437
- System catalog
 - database entries 2-89
 - dropping tables 2-281
 - syscolauth 2-437
 - systabauth 2-358, 2-437
- System descriptor area
 - assigning values to 2-545
 - item descriptors in 2-7
 - modifying contents 2-545
 - resizing 2-549
 - use of 2-264
 - use with EXECUTE statement 2-292
- System name, in database name 4-23
- SYSTEM statement
 - setting environment variables with 3-45
 - syntax 3-44

T

- Table
 - adding a constraint 2-59, 2-60, 2-61
 - alias in SELECT 2-467
 - consumed 2-12
 - creating a synonym for 2-151
 - defining fragmentation strategy 2-181
 - defining temporary 2-194
 - diagnostic 2-607
 - dropping 2-279
 - dropping a synonym 2-277
 - dropping a system table 2-281
 - engine response to locked table 2-562
 - external 2-92, 2-498
 - joins in Condition segment 2-479
 - loading data with the LOAD statement 2-384

- locking
 - changing mode 2-65
 - with ALTER INDEX 2-35
 - with LOCK TABLE 2-391
- logging 2-191
- name, specifying 4-25
- naming conventions 2-156
- optimizing queries 2-637
- resident in memory 2-576
- scratch 2-191, 2-498
- surviving 2-12
- target 2-600
- temporary 2-190, 2-497
- types 2-177
- unlocking 2-621
- violations 2-599
- TABLE keyword, syntax in UPDATE STATISTICS 2-635
- Table Name. *See* Database Object Name.
- Table-level constraints
 - in ALTER TABLE 2-61
- Table-level privilege
 - default with GRANT 2-356
 - granting 2-353
 - passing grant ability 2-357
 - revoking 2-436
- Table-mode format, in SET Database Object Mode statement 2-514
- TABLES keyword, in INFO statement 2-370
- TAN function 4-81
- Target table
 - relationship to diagnostics table 2-600
 - relationship to violations table 2-600
- Temporary table
 - and fragmentation 2-191
 - building distributions 2-644
 - created with SELECT INTO TEMP 2-497
 - creating constraints for 2-195
 - DBSPACETEMP environment variable 2-193
 - defining columns 2-194
 - naming 2-192
 - storage of 2-192

- updating statistics 2-637
- when deleted 2-192
- TEXT data type
 - in LOAD statement 2-388
 - in UNLOAD statement 2-618
 - storage of 2-180
 - syntax 4-27, 4-29
 - with stored procedures 3-9, 3-15
- Thread
 - defined 2-509
 - in multithreaded
 - application 2-509
- Thread-safe application
 - description 2-269, 2-509, 2-511
- Time function
 - restrictions with GROUP BY 2-483
 - use in SELECT 2-459
- to 2-42
- TO CLUSTER keywords, in ALTER INDEX 2-34
- TO keyword
 - in expression 4-98
 - in GRANT 2-344
- TODAY function
 - in ALTER TABLE 2-43
 - in Condition segment 4-6
 - in expression 4-48, 4-52
 - in INSERT 2-378, 2-380
 - using as default value 2-162
- TO_CHAR function 4-79
- TO_DATE function 4-80
- TP/XA. *See* Transaction manager.
- TRACE statement, syntax 3-47
- Transaction
 - description of 2-69
 - example with data manipulation statements 2-69
 - in active connection 2-81
 - in stored procedure 2-143
 - using cursors in 2-255
- Transaction mode,
 - constraints 2-591
- Trigger
 - affected by dropping a column from table 2-55
 - database object modes for setting 2-513

- effects of ALTER FRAGMENT ATTACH 2-13
- effects on altered fragments 2-19
 - in client/server environment 2-226
 - name, specifying 4-25
 - number on a table 2-200
 - overriding 2-226
- Trigger event
 - definition of 2-200
 - in CREATE TRIGGER statement 2-200
 - INSERT 2-207
 - privileges on 2-201
 - with cursor statement 2-201
- Triggered action
 - action on triggering table 2-217
 - action statements 2-212
 - anyone can use 2-223
 - cascading 2-206
 - correlation name in 2-215, 2-222
 - for multiple triggers 2-205
 - list syntax 2-211
 - merged 2-205
 - sequence of 2-204
 - WHEN condition 2-211
- Triggering statement
 - affecting multiple rows 2-206
 - consistent results 2-212
 - execution of 2-201
 - guaranteeing same result 2-200
 - UPDATE 2-203
- Triggering table
 - action on 2-217
 - and cascading triggers 2-225
- Trigonometric function
 - ACOS function 4-83
 - ASIN function 4-83
 - ATAN function 4-83
 - ATAN2 function 4-83
 - COS function 4-82
 - SIN function 4-82
 - TAN function 4-82
- TRIM function 4-85
- TRUNC function 4-56, 4-61
- TYPE field
 - setting in SET DESCRIPTOR 2-549

- setting in X/Open
 - programs 2-550
- with X/Open programs 2-322

U

- Underscore (_), as wildcard 4-12
- UNION operator
 - in SELECT 2-451, 2-500
 - restrictions on use 2-501
- Unique constraint
 - dropping 2-64
 - rules of use 2-166, 2-175
- UNIQUE keyword
 - in aggregate expression 4-98
 - in CREATE INDEX 2-108
 - in SELECT 2-453, 2-457
 - in subquery 4-16
 - with constraint definition 2-175
- UNITS keyword 4-48, 4-54
- UNKNOWN truth values 4-6
- UNLOAD statement
 - DELIMITER clause 2-619
 - syntax 2-616
 - UNLOAD TO file 2-617
 - VARCHAR, TEXT, or BYTE columns 2-618
- UNLOAD TO file 2-617
- UNLOCK TABLE statement, syntax and use 2-621
- Updatable view 2-235
- UPDATE clause, in CREATE TRIGGER 2-202
- Update cursor 2-249
 - locking considerations 2-249
 - opening 2-395
 - restricted statements 2-252
 - use in UPDATE 2-633
- UPDATE keyword
 - in GRANT 2-353
 - in REVOKE 2-436
 - in REVOKE FRAGMENT statement 2-446
- Update privilege, with a view 2-625
- UPDATE statement
 - and transactions 2-626
 - and triggers 2-200, 2-202, 2-212

- locking considerations 2-627
- restrictions on columns for
 - update 2-250
- rolling back updates 2-627
- syntax 2-624
- updating a column to null 2-630
- updating a column twice 2-632
- updating through a view 2-625
- updating with cursor 2-633
- use of expressions 2-631
- with
 - FETCH 2-309
 - SELECT...FOR UPDATE 2-492
 - SET keyword 2-628
 - update cursor 2-249
 - WHERE CURRENT OF
 - keywords 2-633
 - WHERE keyword 2-632
- UPDATE STATISTICS statement
 - and temporary tables 2-644
 - creating data distributions 2-641
 - dropping data distributions 2-640
 - examining index pages 2-638
 - optimizing search
 - strategies 2-637
 - specifying distributions
 - only 2-644
 - syntax 2-635
 - using
 - after modifying tables 2-638
 - the LOW keyword 2-640
 - when upgrading database
 - server 2-639
- Update trigger, defining
 - multiple 2-203
- Upgrading the database
 - server 2-639
- Upgrading to a newer database
 - server 2-639
- USER function
 - as affected by ANSI
 - compliance 2-434, 4-51
 - in ALTER TABLE 2-43
 - in Condition segment 4-6
 - in expression 4-48, 4-50
 - in INSERT 2-378, 2-380
 - using as default value 2-162
- User informix, privileges associated
 - with 2-348

- USING BITMAP keywords, in
 - CREATE INDEX 2-107
- Using correlation names 2-214
- USING DESCRIPTOR keywords
 - in DESCRIBE 2-264
 - in FETCH 2-309
 - in OPEN 2-394, 2-400
 - in PUT 2-292, 2-422
 - syntax in EXECUTE 2-290
 - use
 - in PUT 2-423
- USING keyword
 - in EXECUTE 2-290
 - in OPEN 2-394, 2-398
- USING SQL DESCRIPTOR
 - keywords
 - in DESCRIBE 2-265
 - in EXECUTE 2-292

V

- VALUE clause
 - after NULL value is fetched 2-323
 - relation to FETCH 2-323
 - use in GET DESCRIPTOR 2-322
 - use in SET DESCRIPTOR 2-549
- VALUES clause
 - effect with PUT 2-421
 - in INSERT 2-378
 - syntax in INSERT 2-373
- VARCHAR data type
 - in LOAD statement 2-388
 - in UNLOAD statement 2-618
- syntax 4-29
- using as default value 2-161
- Variable
 - default values in SPL 3-12, 3-14
 - define in SPL 3-8
 - global, in SPL 3-10
 - local, in SPL 3-14
 - scope of SPL variable 3-9
 - unknown values in IF 3-28
- VARIANCE function 4-108
- View
 - affected by dropping a
 - column 2-55
 - creating a view 2-230
 - creating synonym for 2-151

- dropping 2-283
- name, specifying 4-25
- privileges with GRANT 2-359
- union 2-232
- updatable 2-235
- updating 2-625
 - with SELECT * notation 2-231
- View Name. *See* Database Object Name.
- Violations table
 - creating with START
 - VIOLATIONS TABLE 2-595
 - examples 2-521, 2-527, 2-604, 2-607, 2-615
 - how to stop 2-614
 - privileges on 2-601
 - relationship to diagnostics
 - table 2-600
 - relationship to target table 2-600
 - restriction on dropping 2-281
 - structure 2-599
 - use with SET Database Object
 - Mode statement 2-518
 - when to start 2-518

W

- WAIT keyword, in the SET LOCK
 - MODE statement 2-562
- Warning, if dbspace skipped 2-535
- WEEKDAY function 4-73, 4-76
- WHEN condition, in triggers 2-211
- WHenever statement
 - CALL keyword 2-646
 - syntax and use 2-646
 - with concatenation operator 4-36
- WHERE clause
 - in SELECT 2-451
 - joining tables 2-479
 - with a subquery 2-473
 - with ALL keyword 2-478
 - with ANY keyword 2-478
 - with BETWEEN keyword 2-474
 - with IN keyword 2-474
 - with IS keyword 2-475
 - with relational operator 2-473
 - with SOME keyword 2-478

WHERE CURRENT OF keywords
 impact on trigger 2-201
 in DELETE 2-258
 in UPDATE 2-623, 2-632

WHERE keyword
 in DELETE 2-258, 2-260
 in UPDATE 2-623, 2-632
 setting descriptions of
 items 2-545

WHILE keyword
 in CONTINUE statement 3-7
 in EXIT 3-16

WHILE statement
 syntax 3-50
 with NULL expressions 3-50

Wildcard character
 asterisk (*) 4-13
 backslash (\) 4-12, 4-13
 caret (^) 4-13
 percent sign (%) 4-12
 question mark (?) 4-13
 square brackets [...] 4-13
 underscore (_) 4-12
 with LIKE 2-475, 4-12
 with LIKE or MATCHES 4-160
 with MATCHES 2-475, 4-13

wing 4-87

WITH APPEND keywords, in the
 SET DEBUG FILE TO
 statement 2-538

WITH BUFFERED LOG keywords,
 in CREATE DATABASE 2-90

WITH CHECK keywords, in
 CREATE VIEW 2-230, 2-234

WITH CRCOLS keywords, in
 CREATE TABLE 2-178

WITH GRANT keywords, syntax in
 GRANT 2-344

WITH GRANT OPTION
 in GRANT 2-357
 in REVOKE 2-434

WITH HOLD keywords, in
 DECLARE 2-241, 2-247, 2-257

WITH LOG MODE ANSI
 keywords, in CREATE
 DATABASE 2-90

WITH NO CHECK OPTION, in
 ALTER FRAGMENT 2-14

WITH NO LOG keywords
 in CREATE TABLE 2-191
 in SELECT 2-497, 2-498

WITH RESUME keywords, in
 RETURN 3-42

WITHOUT HEADINGS keywords,
 in OUTPUT 2-403

X

X/Open compliance
 level Intro-20

X/Open mode
 FETCH statement 2-303
 SET DESCRIPTOR
 statement 2-550

Y

YEAR function 4-73, 4-77

YEAR keyword
 in DATETIME data type 4-32
 in DATETIME segment 4-134
 in INTERVAL 4-131, 4-137

Z

Zoned decimal, in external
 tables 2-94

Symbols

", double quotes
 around quoted string 4-158
 in quoted string 4-159
 with delimited identifiers 4-116,
 4-119

\$INFORMIXDIR /etc/sqlhosts. *See*
 sqlhosts file.

%, percent sign, as wildcard 4-12

(|), pipe character 2-100, 2-499

*, asterisk
 as wildcard character 4-13

*, asterisk
 arithmetic operator 4-34
 use in SELECT 2-453

+, plus sign, arithmetic
 operator 4-34

--, double dash, comment
 symbol 1-6

-, minus sign, arithmetic
 operator 4-34

., period 2-135

/, division symbol, arithmetic
 operator 4-34

?, question mark
 as placeholder in PREPARE 2-405
 as wildcard 4-13
 naming variables in PUT 2-422
 replacing with USING
 keyword 2-398

[...], square brackets
 as wildcard characters 4-13

\, backslash, as wildcard
 character 4-12, 4-13

^, caret, as wildcard character 4-13

_, underscore, as wildcard 4-12

{}, curly brackets, comment
 symbol 1-6

| |, concatenation operator 4-34,
 4-35

', single quotes
 around quoted string 4-158
 in quoted string 4-159