

IBM Informix Guide to SQL

Reference

IBM Informix 4GL, Version 4.1
IBM Informix SQL, Version 4.1
IBM Informix ESQL/C, Version 5.2
IBM Informix ESQL/COBOL, Version 5.0
IBM Informix SE, Version 5.0
IBM Informix OnLine, Version 5.2
IBM Informix NET, Version 5.2
IBM Informix STAR, Version 5.2

November 2002
Part No. 000-9122

Note:

Before using this information and the product it supports, read the information in the appendix entitled “Notices.”

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 2002. All rights reserved.

US Government User Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Table of Contents

Introduction

In This Introduction	3
About This Manual	3
Organization of This Manual	4
IBM Informix Products That Use SQL	5
Products Covered in This Manual	5
The Demonstration Database	6
Creating the Demonstration Database on IBM Informix OnLine	7
Creating the Demonstration Database on IBM Informix SE.	8
New Features in IBM Informix Server Products, Version 5.x	9
Document Conventions	11
Typographical Conventions	11
Syntax Conventions	12
Example Code Conventions	17
Additional Documentation	18
Online Manuals	18
Error Message Files	19
Documentation Notes, Release Notes, Machine Notes	22
Compliance with Industry Standards	22
IBM Welcomes Your Comments	23

Chapter 1

The stores5 Database

In This Chapter	1-3
Structure of the Tables	1-4
The customer Table	1-5
The orders Table	1-6
The items Table	1-6
The stock Table	1-8
The catalog Table	1-9
The cust_calls Table	1-10
The call_type Table	1-10
The manufact Table	1-11
The state Table.	1-11
The stores5 Database Map	1-11
Primary-Foreign Key Relationships	1-13
The customer and orders Tables.	1-14
The orders and items Tables	1-15
The items and stock Tables	1-16
The stock and catalog Tables	1-17
The stock and manufact Tables	1-18
The cust_calls and customer Tables	1-19
The call_type and cust_calls Table	1-20
The state and customer Tables	1-21
Data in the stores5 Database	1-21

Chapter 2

System Catalog

In This Chapter	2-3
Using the System Catalog	2-4
Accessing the System Catalog	2-8
Updating System Catalog Data	2-9
Structure of the System Catalog	2-9
SYSBLOBS	2-10
SYSCHECKS	2-11
SYSCOLAUTH	2-11
SYSCOLDEPEND	2-12
SYSCOLUMNS	2-13
SYSCONSTRAINTS.	2-16
SYSDEFAULTS	2-17
SYSDEPEND	2-18
SYSINDEXES	2-18

SYSOPCLSTR	2-21
SYSROCAUTH	2-23
SYSROCBODY	2-24
SYSROCEDURES	2-25
SYSROCPPLAN	2-26
SYSREFERENCES	2-27
SYSNONYMS	2-27
SYSYNTABLE	2-28
SYSTABAUTH	2-29
SYSTABLES	2-30
SYSUSERS	2-32
SYSVIEWS	2-33
System Catalog Map	2-33

Chapter 3 Data Types

In This Chapter	3-3
Database Data Types	3-4
BYTE	3-5
CHAR(n)	3-6
CHARACTER(n)	3-7
DATE	3-7
DATETIME	3-8
DEC.	3-11
DECIMAL[(p,s)]	3-11
DOUBLE PRECISION(n)	3-12
FLOAT(n).	3-12
INT	3-13
INTEGER.	3-13
INTERVAL	3-13
MONEY(p,s).	3-17
NUMERIC(p,s)	3-17
REAL	3-17
SERIAL(n)	3-18
SMALLFLOAT	3-19
SMALLINT	3-19
TEXT	3-19
VARCHAR(m,r)	3-21

Data Type Conversions	3-22
Converting from Number to Number.	3-23
Converting Between Number and CHAR	3-24
Converting Between DATE and DATETIME	3-24
Range of Operations Using DATE, DATETIME, and INTERVAL	3-25
Manipulating DATETIME Values	3-26
Manipulating DATETIME with INTERVAL Values	3-27
Manipulating DATE with DATETIME and INTERVAL Values	3-28
Manipulating INTERVAL Values	3-30
Multiplying or Dividing INTERVAL Values	3-30

Chapter 4 Environment Variables

In This Chapter	4-3
Setting Environment Variables	4-4
Informix Environment Variables	4-5
DBANSIWARN	4-7
DBDATE	4-8
DBDELIMITER	4-9
DBEDIT	4-10
DBFORMAT	4-10
DBLANG	4-11
DBMENU	4-12
DBMONEY.	4-12
DBNETTYPE	4-13
DBPATH.	4-14
DBPRINT	4-15
DBREMOTECMD	4-15
DBSRC	4-16
DBTEMP	4-17
DBTIME.	4-17
INFORMIXCOB	4-20
INFORMIXCOBDIR.	4-20
INFORMIXCOBSTORE	4-21
INFORMIXCOBTYPE	4-21
INFORMIXDIR	4-22
INFORMIXONLINEDIR	4-23
INFORMIXTERM	4-23
NOSORTINDEX	4-24
SQLEXEC	4-25
SQLRM	4-26

	SQLRMDIR	4-27
	TBCONFIG	4-27
	UNIX Environment Variables	4-28
	PATH	4-28
	TERM	4-29
	TERMCAP	4-29
	TERMINFO	4-30
Chapter 5	Error Handling with SQLCA	
	In This Chapter	5-3
	The SQLCA Record in IBM Informix 4GL	5-5
	The sqlca Structure in IBM Informix ESQL/C	5-7
	The SQLCA Record in IBM Informix ESQL/COBOL	5-10
Chapter 6	Using Descriptors	
	In This Chapter	6-3
	The System Descriptor Area and the sqlda Structure in ESQL/C	6-4
	Using a System Descriptor Area	6-5
	Using Pointers to an sqlda Structure	6-9
	The System Descriptor Area in ESQL/COBOL	6-13
	Using a System Descriptor Area	6-13
Chapter 7	Syntax	
	In This Chapter	7-9
	SQL Statements	7-9
	ALLOCATE DESCRIPTOR	7-13
	ALTER INDEX	7-17
	ALTER TABLE	7-20
	BEGIN WORK	7-37
	CHECK TABLE	7-39
	CLOSE	7-41
	CLOSE DATABASE	7-44
	COMMIT WORK	7-46
	CREATE AUDIT	7-47
	CREATE DATABASE	7-49
	CREATE INDEX	7-54
	CREATE PROCEDURE	7-58
	CREATE PROCEDURE FROM	7-67
	CREATE SCHEMA	7-68
	CREATE SYNONYM	7-70
	CREATE TABLE	7-75

CREATE VIEW	7-97
DATABASE.	7-101
DEALLOCATE DESCRIPTOR	7-105
DECLARE	7-107
DELETE	7-122
DESCRIBE	7-125
DROP AUDIT	7-131
DROP DATABASE	7-132
DROP INDEX	7-134
DROP PROCEDURE	7-136
DROP SYNONYM	7-137
DROP TABLE	7-139
DROP VIEW	7-141
EXECUTE	7-142
EXECUTE IMMEDIATE	7-147
EXECUTE PROCEDURE	7-150
FETCH	7-153
FLUSH	7-162
FREE	7-165
GET DESCRIPTOR	7-169
GRANT	7-175
INFO	7-185
INSERT	7-189
LOAD	7-199
LOCK TABLE	7-204
OPEN.	7-207
OUTPUT	7-216
PREPARE	7-218
PUT	7-230
RECOVER TABLE	7-238
RENAME COLUMN	7-241
RENAME TABLE	7-243
REPAIR TABLE	7-245
REVOKE	7-247
ROLLBACK WORK.	7-254
ROLLFORWARD DATABASE	7-256
SELECT	7-258
SET CONSTRAINTS	7-289
SET DEBUG FILE TO	7-291
SET DESCRIPTOR	7-293
SET EXPLAIN.	7-301
SET ISOLATION	7-307

SET LOCK MODE	7-311
SET LOG	7-313
SET OPTIMIZATION	7-315
START DATABASE	7-317
UNLOAD	7-319
UNLOCK TABLE	7-323
UPDATE	7-325
UPDATE STATISTICS	7-335
WHENEVER	7-337
Segments	7-344
Condition	7-345
Constraint Name	7-360
Database Name	7-362
Data Type	7-365
DATETIME Field Qualifier	7-368
Expression	7-370
Identifier	7-399
Index Name	7-413
INTERVAL Field Qualifier	7-414
Literal DATETIME	7-416
Literal INTERVAL	7-419
Literal Number	7-422
Procedure Name	7-424
Quoted String	7-426
Relational Operator	7-429
Synonym Name	7-432
Table Name	7-434
View Name	7-438

Chapter 8 Stored Procedures and SPL

In This Chapter	8-5
Introduction to Stored Procedures and SPL	8-5
What You Can Do with Stored Procedures	8-6
Relationship Between SQL and a Stored Procedure	8-6
Creating and Using Stored Procedures	8-7
Creating a Procedure Using DB-Access	8-7
Creating a Procedure Using an Embedded-Language Product	8-8
Commenting and Documenting a Procedure	8-8
Diagnosing Compile-Time Errors	8-9
Looking at Compile-Time Warnings	8-10
Generating the Text or Documentation	8-11
Executing a Procedure	8-12

Debugging a Procedure	8-14
Re-creating a Procedure	8-16
Privileges on Stored Procedures	8-16
Privileges at Creation	8-16
Privileges at Execution	8-17
Revoking Privileges	8-19
Variables and Expressions	8-19
Variables.	8-19
Expressions.	8-23
Program Flow Control	8-26
Branching	8-26
Looping	8-27
Function Handling	8-28
Passing Information into and out of a Procedure	8-29
Returning Results	8-29
Exception Handling	8-32
Trapping an Error and Recovering	8-32
Scope of Control of an ON EXCEPTION Statement	8-33
User-Generated Exceptions	8-34
SPL Statement Syntax	8-36
CALL.	8-37
CONTINUE	8-40
DEFINE	8-42
EXIT	8-50
FOR	8-52
FOREACH	8-56
IF	8-60
LET	8-64
ON EXCEPTION.	8-67
RAISE EXCEPTION.	8-73
RETURN	8-75
SYSTEM	8-78
TRACE	8-80
WHILE	8-84

Appendix A Notices

Glossary

Index

Introduction

In This Introduction	3
About This Manual.	3
Organization of This Manual	4
IBM Informix Products That Use SQL	5
Products Covered in This Manual.	5
The Demonstration Database	6
Creating the Demonstration Database on IBM Informix OnLine	7
Creating the Demonstration Database on IBM Informix SE.	8
New Features in IBM Informix Server Products, Version 5.x.	9
Document Conventions	11
Typographical Conventions	11
Syntax Conventions	12
Example Code Conventions	17
Additional Documentation	18
Online Manuals	18
Error Message Files	19
The finderr Script.	20
The rofferr Script	21
Documentation Notes, Release Notes, Machine Notes	22
Compliance with Industry Standards	22
IBM Welcomes Your Comments	23

In This Introduction

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

About This Manual

IBM Informix Guide to SQL: Reference is intended to be used as a companion volume to *IBM Informix Guide to SQL: Tutorial*. Like *IBM Informix Guide to SQL: Tutorial*, this book is written for people who already know how to use computers and who rely on them in their daily work.

Whereas *IBM Informix Guide to SQL: Tutorial* explains the philosophy and concepts behind relational databases, this volume is a reference source that you can use on a daily basis after you have finished reading and experimenting with *IBM Informix Guide to SQL: Tutorial*.

Organization of This Manual

IBM Informix Guide to SQL: Reference includes the following chapters:

- [Chapter 1, “The stores5 Database,”](#) describes the structure and contents of the demonstration database named **stores5** that is installed with every IBM Informix application development tool. It includes a map of the nine tables in the database, illustrates the columns on which they join, and displays the data in them.
- [Chapter 2, “System Catalog,”](#) provides details of the Informix system catalog, which is the collection of 21 system catalog tables that describe the structure of the **stores5** database. The chapter explains how to access and update statistics in the system catalog, shows the system catalog structure, and lists the name and data type for each column in each table.
- [Chapter 3, “Data Types,”](#) defines the column data types supported by IBM Informix products, tells how to convert between different data types, and describes how to use specific values in arithmetic and relational expressions.
- [Chapter 4, “Environment Variables,”](#) describes the various environment variables that you can or should set to properly use your IBM Informix products. These variables identify your terminal, specify the location of your software, and define other parameters of your product environment.
- [Chapter 5, “Error Handling with SQLCA,”](#) explains how errors are handled and tells how you can check the contents of the SQL Communications Area (SQLCA) when you use the following products: IBM Informix 4GL, IBM Informix ESQL/C, or IBM Informix ESQL/COBOL.
- [Chapter 6, “Using Descriptors,”](#) describes the system descriptor area and the SQL Descriptor Area (**sqllda**), which hold descriptive information about data in dynamic SQL statements.
- [Chapter 7, “Syntax,”](#) explains the workings of all the SQL statements supported by IBM Informix products. Detailed diagrams walk you through every clause of each SQL statement. Thorough usage instructions, pertinent examples, and references to related material complete the SQL picture.

- [Chapter 8, “Stored Procedures and SPL,”](#) describes how to create and use stored procedures. It also contains the syntax of the Stored Procedure Language (SPL) statements.
- A Notices appendix describes IBM products, features, and services.
- A glossary of common database terms follows the chapters, and a comprehensive index directs you to areas of particular interest.

IBM Informix Products That Use SQL

IBM produces many application development tools and CASE tools that use SQL. Application development tools currently available include products like IBM Informix SQL, IBM Informix 4GL and the IBM Informix 4GL Interactive Debugger, and the IBM Informix embedded-language products, such as IBM Informix ESQL/C.

IBM Informix products work with a database server, either IBM Informix OnLine or IBM Informix SE. If you are running applications on a network, you will use an IBM Informix client/server product such as IBM Informix NET or IBM Informix STAR. IBM Informix NET is the communication facility for multiple IBM Informix SE database servers. IBM Informix STAR allows distributed database access to multiple IBM Informix OnLine database servers. You also can use IBM Informix NET on a client to access remote OnLine database servers (as long as IBM Informix STAR is installed with OnLine on the same database server).

Products Covered in This Manual

The information presented in this manual is valid for the following products and versions, and indicates differences in their use of SQL where appropriate:

- IBM Informix 4GL (C Compiler Version and Rapid Development System Version) Version 4.1
- IBM Informix SQL Version 4.1
- IBM Informix ESQL/C Version 5.2
- IBM Informix ESQL/COBOL Version 5.0

- IBM Informix SE Version 5.0
- IBM Informix NET Version 5.2
- IBM Informix OnLine Version 5.2
- IBM Informix STAR Version 5.2

The *IBM Informix TP/XA User Manual* discusses the special considerations you should be aware of when using SQL statements with IBM Informix TP/XA.

The Demonstration Database

The DB-Access utility, which is provided with your IBM Informix database server products, includes a demonstration database called **stores5** that contains information about a fictitious wholesale sporting-goods distributor. The sample command files that make up a demonstration application are included as well.

Most of the examples in this manual are based on the **stores5** demonstration database. The **stores5** database is described in detail and its contents are listed in [Chapter 1, “The stores5 Database.”](#)

The script that you use to install the demonstration database is called **dbaccessdemo5** and is located in the **\$INFORMIXDIR/bin** directory. The database name that you supply is the name given to the demonstration database. If you do not supply a database name, the name defaults to **stores5**. Follow these rules for naming your database:

- Names for databases can be up to 10 characters long.
- The first character of a name must be a letter.
- You can use letters, characters, and underscores (_) for the rest of the name.
- DB-Access makes no distinction between uppercase and lowercase letters.
- The database name should be unique.

When you run **dbaccessdemo5**, you are, as the creator of the database, the owner and Database Administrator (DBA) of that database.

If you installed your IBM Informix database server product according to the installation instructions, the files that make up the demonstration database are protected so that you cannot make any changes to the original database.

You can run the **dbaccessdemo5** script again whenever you want to work with a fresh demonstration database. The script prompts you when the creation of the database is complete, and asks if you would like to copy the sample command files to the current directory. Answer “N” to the prompt if you have made changes to the sample files and do not want them replaced with the original versions. Answer “Y” to the prompt if you want to copy over the sample command files.

Creating the Demonstration Database on IBM Informix OnLine

Use the following steps to create and populate the demonstration database in the IBM Informix OnLine environment:

1. Set the INFORMIXDIR environment so that it contains the name of the directory in which your IBM Informix products are installed. Set SQLEXEC to **\$INFORMIXDIR/lib/sqlturbo**. (For a full description of environment variables, see [Chapter 4, “Environment Variables.”](#))
2. Create a new directory for the SQL command files. Create the directory by entering

```
mkdir dirname
```

3. Make the new directory the current directory by entering
4. Create the demonstration database and copy over the sample command files by entering

```
dbaccessdemo5 dbname
```

The data for the database is put into the root dbspace.

To give someone else the SQL privileges to access the data, use the GRANT and REVOKE statements. The GRANT and REVOKE statements are described in [Chapter 7](#).

To use the command files that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **dbaccessdemo5** script. To give someone else the permissions to access the command files in your directory, use the UNIX **chmod** command.

Creating the Demonstration Database on IBM Informix SE

Use the following steps to create and populate the demonstration database in the IBM Informix SE environment:

1. Set the **INFORMIXDIR** environment so that it contains the name of the directory in which your IBM Informix products are installed. Set **SQLEXEC** to **\$INFORMIXDIR/lib/sqlexec**. (For a full description of environment variables, see [Chapter 4](#).)
2. Create a new directory for the demonstration database. This directory will contain the example command files included with the demonstration database. Create the directory by entering

```
mkdir dirname
```
3. Make the new directory the current directory by entering

```
cd dirname
```
4. Create the demonstration database and copy over the sample command files by entering

```
dbaccessdemo5 dbname
```

When you run the **dbaccessdemo5** script, it creates a subdirectory called **dbname.dbs** in your current directory and places the database files associated with **stores5** there. You will see both data and index files in the **dbname.dbs** directory.

To use the database and the command files that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **dbaccessdemo5** script. To give someone else the permissions to access the command files in your directory, use the UNIX **chmod** command. Check with your system administrator for more information about operating system file and directory permissions. UNIX permissions are discussed in the *IBM Informix SE Administrator's Guide*.

To give someone else access to the database that you have created, grant them the appropriate privileges using the GRANT statement in DB-Access. To remove privileges, use the REVOKE statement. The GRANT and REVOKE statements are described in [Chapter 7](#).

New Features in IBM Informix Server Products, Version 5.x

This section highlights the major new features implemented in version 5.x of IBM Informix server products:

- **Enhanced Connectivity (IBM Informix OnLine only)**

The version 5.2 IBM Informix OnLine database server enables you to connect to Version 7.x client application tools when both server and client are installed in the same machine.
- **Enhanced support for chunk offsets (IBM Informix OnLine only)**

The version 5.2 IBM Informix OnLine database server supports chunk offset values up to 2 terabytes.
- **Referential and Entity Integrity**

New data integrity constraints allow you to specify a column or columns as representing a *primary* or *foreign key* of a table upon creation, and to establish dependencies between tables. Once specified, a parent-child relationship between two tables is enforced by the database server. Other constraints allow you to specify a default value for a column, or to specify a condition for a column that an inserted value must meet.
- **Stored Procedures**

A stored procedure is a function written by a user using a combination of SQL statements and Stored Procedure Language (SPL). Once created, a procedure is stored as an object in the database in a compiled, optimized form, and is available to other users with the appropriate privileges. In a client/server environment, the use of stored procedures can significantly reduce network traffic.

- **Dynamic SQL**

Support is provided for the X/Open implementation of dynamic SQL using a system descriptor area. This support involves the new SQL statements `ALLOCATE DESCRIPTOR`, `DEALLOCATE DESCRIPTOR`, `GET DESCRIPTOR`, and `SET DESCRIPTOR`, as well as changes in the syntax of existing dynamic management statements.

- **Optimizer Enhancement**

You can use the new `SET OPTIMIZATION` statement to instruct the database server to select a high or low level of query optimization. The default level of `HIGH` causes the database server to examine and select the best of all possible optimization strategies. Since this level of optimization may result in a longer-than-desired optimization time for some queries, you have the option of setting an optimization level of `LOW`.

- **Relay Module (IBM Informix NET only)**

The new Relay Module component of IBM Informix NET resides on the client machine in a distributed data processing environment and *relays* messages between an application development tool and an IBM Informix OnLine or IBM Informix SE database server through a network interface. The Relay Module allows version 5.0 application development tools to connect to a remote database server without the need to run an Informix database server process on the client.

- **Two-Phase Commit (IBM Informix STAR only)**

The new two-phase commit protocol allows you to manipulate data in multiple databases on multiple OnLine database servers within a single transaction. It ensures that transactions that span more than one OnLine database server are committed on an all-or-nothing basis.

- **Support for Transaction Processing in the XA Environment (IBM Informix TP/XA only)**


IBM Informix TP/XA allows you to use the OnLine database server as a Resource Manager in conformance with the *X/Open Preliminary Specification (April 1990), Distributed Transaction Processing: The XA Interface*. The *IBM Informix TP/XA User Manual* describes the changes in the behavior of existing SQL statements that manage transactions in an X/Open environment.

Document Conventions

This manual assumes that you are using IBM Informix OnLine as your database server. Features and behavior specific to IBM Informix SE are noted throughout the manual.

Typographical Conventions

IBM Informix product manuals use a standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth. The following typographical conventions are used throughout this manual:

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
<i>italics</i>	Within text, new terms and emphasized words appear in italics.
<i>italics</i>	Within syntax and code examples, variable values that you are to specify appear in italics.
boldface	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
<i>boldface</i>	
monospace <i>monospace</i>	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
◆	This symbol indicates the end of product- or platform-specific information.
	Indicates a unique identifier (primary key) for each table.



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

Syntax Conventions

Syntax diagrams describe the format of SQL statements or commands, including alternative forms of a statement, required and optional parts of the statement, and so forth. Syntax diagrams have their own conventions, which are defined in detail and illustrated in this section. SQL statements are listed in their entirety in [Chapter 7, “Syntax,”](#) although some statements may appear in other manuals.

Each syntax diagram displays the sequences of required and optional elements that are valid in a statement. Briefly:

- All keywords are shown in uppercase letters for ease of identification, even though you need not enter them that way.
- Words for which you must supply values are in italics.

A diagram begins at the upper left with a keyword. It ends at the upper right with a vertical line. Between these points you can trace any path that does not stop or back up. Each path describes a valid form of the statement.





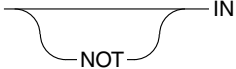
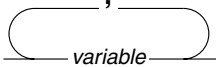
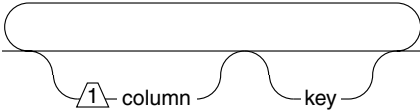
Along a path, you may encounter the following elements:

Element	Description
KEYWORD	You must spell a word in uppercase letters exactly as shown; however, you can use either uppercase or lowercase letters when you enter it.
(.,;+*-/)	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
" "	Double quotes are literal symbols that you must enter as shown. You can replace a pair of double quotes with a pair of single quotes, if you prefer. You cannot mix double and single quotes.
<i>variable</i>	A word in italics represents a value that you must supply. The nature of the value is explained immediately following the diagram unless the variable appears in a box. In that case, the page number of the detailed explanation follows the variable name.

(1 of 3)

Element	Description
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> ADD Clause p. 7-14 </div>	<p>A reference in a box represents a subdiagram on the same page or another page. Imagine that the subdiagram is spliced into the main diagram at this point.</p>
<div style="border: 1px solid black; padding: 2px; display: inline-block;"> Relational Operator <i>see</i> SQLR </div>	<p>A reference to SQLR in another manual represents an SQL statement or segment described in Chapter 7, “Syntax.” Imagine that the statement or segment is spliced into the main diagram at this point.</p>
I4GL	<p>A code in an icon is a signal warning you that this path is valid only for some products or under certain conditions. The codes indicate the products or conditions that support the path. The following codes are used:</p>
SE	This path is valid only for IBM Informix SE.
OL	This path is valid only for IBM Informix OnLine.
STAR	This path is valid only for IBM Informix STAR.
INET	This path is valid only for IBM Informix NET.
I4GL	This path is valid only for IBM Informix 4GL.
ISQL	This path is valid only for IBM Informix SQL.
ESQL	<p>This path is valid for SQL statements in all the following embedded-language products: IBM Informix ESQL/C, or IBM Informix ESQL/COBOL.</p>
E/C	This path is valid only for IBM Informix ESQL/C.
E/CO	This path is valid only for IBM Informix ESQL/COBOL.
DB	This path is valid only for DB-Access.
SPL	This path is valid only if you are using Informix Stored Procedure Language (SPL).

(2 of 3)

Element	Description
	This path is an Informix extension to ANSI-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, you receive the warnings at run time. To receive the warnings at compile time, compile with the -ansi flag.
	A shaded option is the default. Even if you do not explicitly type the option, it will be in effect unless you choose another option.
	Syntax enclosed in a pair of arrows indicates that this is a subdiagram.
	The vertical line is a terminator and indicates that the statement is complete.
	A branch below the main line indicates an optional path.
	A loop indicates a path that can be repeated.
	A gate ($\sqrt{1}$) in an option indicates that you can only use that option once, even though it is within a larger loop.

(3 of 3)

In [Chapter 7](#), icons that appear in the left margin indicate that the accompanying shaded text is valid only for some products or under certain conditions. In addition to the icons described in the preceding list, you may encounter the following icons in the left margin:

ANSI

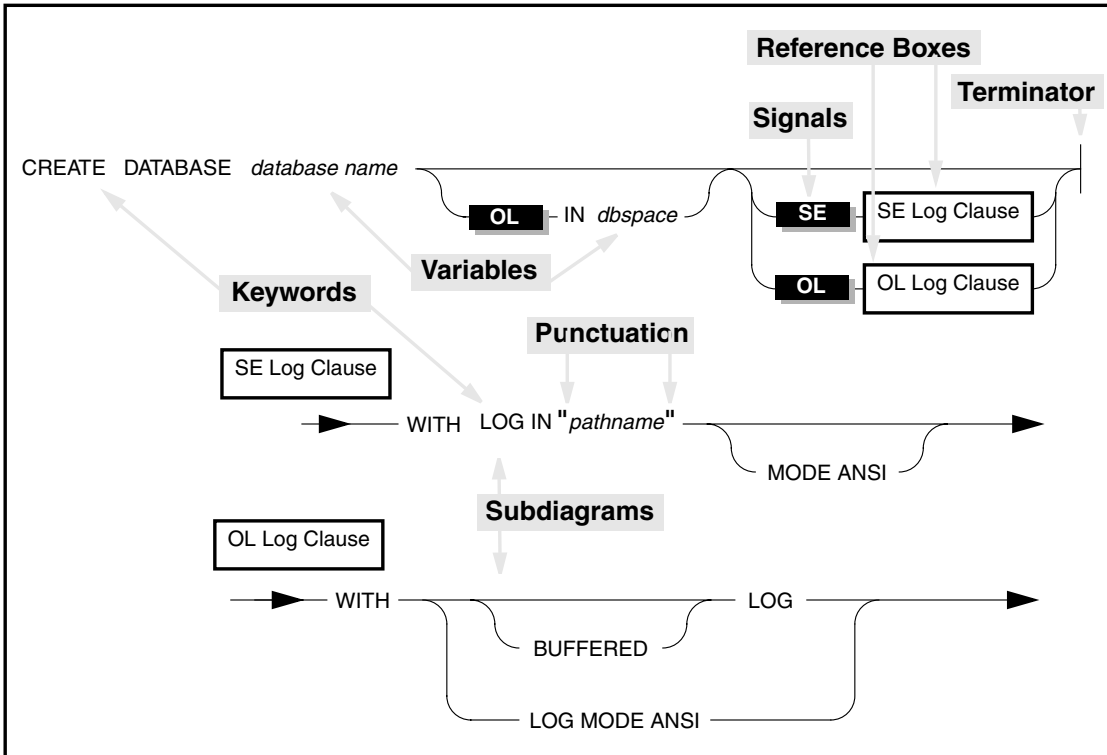
This icon indicates that the functionality described in the shaded text is valid only if your database is ANSI-compliant.

X/O

This icon indicates that the functionality described in the shaded text conforms to X/Open standards for dynamic SQL. This functionality is available when you compile your embedded-language application with the **-xopen** flag.

[Figure 1](#) shows the elements of a syntax diagram for the CREATE DATABASE statement.

Figure 1
Elements of a syntax diagram



To construct a statement using this diagram, start at the top left with the keywords `CREATE DATABASE`. Then follow the diagram to the right, proceeding through the options that you want. The diagram conveys the following information:

1. You must type the words `CREATE DATABASE`.
2. You must supply a *database name*.
3. You can stop, taking the direct route to the terminator, or you can take one or more of the optional paths.
4. If desired, you can designate a *dbspace* by typing the word `IN` and a *dbspace name*.

5. If desired, you can specify logging. Here, you are constrained by the database server with which you are working.
 - If you are using IBM Informix OnLine, go to the subdiagram named *OL Log Clause*. Follow the subdiagram by typing the keyword WITH, then choosing and typing either LOG, BUFFERED LOG, or LOG MODE ANSI. Then, follow the arrow back to the main diagram.
 - If you are using IBM Informix SE, go to the subdiagram named *SE Log Clause*. Follow the subdiagram by typing the keywords WITH LOG IN, typing a double quote, supplying a pathname, and closing the quotes. You can then choose the MODE ANSI option below the line or continue to follow the line across.
6. Once you are back at the main diagram, you come to the terminator. Your CREATE DATABASE statement is complete.

Example Code Conventions

Examples of SQL code appear throughout this manual. Except where noted, the code is not specific to any single IBM Informix application development tool. If only SQL statements are listed, they are not delineated by semicolons. To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using DB-Access or IBM Informix SQL, you must delineate the statements with semicolons. If you are using an embedded language, you must use EXEC SQL and a semicolon (or other appropriate delimiters) at the start and end of each statement, respectively.

For example, you might see the following example code:

```

DATABASE stores
.
.
.
DELETE FROM customer
  WHERE customer_num = 121
.
.
.
COMMIT WORK
CLOSE DATABASE

```

If you are using DB-Access or IBM Informix SQL, add semicolons at the end of each statement. If you are using IBM Informix 4GL, use the code as it appears. If you are using IBM Informix ESQL/C, add EXEC SQL or a dollar sign (\$) at the beginning of each line and end each line with a semicolon. For detailed directions on using SQL statements for a particular application development tool, see the manual for your product.

Also note that dots in the 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.

Additional Documentation

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

- Online manuals
- Error message files
- Documentation notes, release notes, and machine notes

Online Manuals

A CD that contains your manuals in electronic format is provided with your IBM Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print online manuals, see the installation insert that accompanies your CD. You can also obtain the same online manuals at the IBM Informix Online Documentation site at <http://www-3.ibm.com/software/data/informix/pubs/library/>.

You may want to refer to a number of related IBM Informix documents that complement *IBM Informix Guide to SQL: Reference*:

- The *SQL Quick Syntax Guide* contains syntax diagrams for all statements and segments described in this manual.
- A companion volume to the Reference, *IBM Informix Guide to SQL: Tutorial*, provides a tutorial on SQL as it is implemented by IBM Informix products. It describes the fundamental ideas and terminology that are used when planning and implementing a relational database. It also describes how to retrieve information from a database, how to modify a database, and how to write a program to retrieve database information and modify a database.
- You, or whoever installs your IBM Informix products, should refer to the *UNIX Products Installation Guide* for your particular release to ensure that your IBM Informix product is properly set up before you begin to work with it.
- If you are using your IBM Informix product across a network, you also may want to refer to the *IBM Informix NET and IBM Informix STAR Installation and Configuration Guide*.
- Depending on the database server you are using, you or your system administrator need either the *IBM Informix OnLine Administrator's Guide* or the *IBM Informix SE Administrator's Guide*.
- When errors occur, you can look them up, by number, and learn their cause and solution in the *IBM Informix Error Messages* manual. If you prefer, you can look up the error messages in the online message file described in the section "[Error Message Files](#)" below.

Error Message Files

IBM Informix software products provide ASCII files that contain all of the error messages and their corrective actions. For a detailed description of these error messages, refer to the *IBM Informix Error Messages* manual in the IBM Informix Online Documentation site at <http://www-3.ibm.com/software/data/informix/pubs/library/>.

In addition, there are two ways in which you can access the error messages directly from the ASCII Error Message File.

- Use the **finderr** script to display one or more error messages on the terminal screen.
- Use the **rofferr** script to print one error message or a range of error messages.

The scripts are in the `$INFORMIXDIR/bin` directory. The ASCII file has the following path:

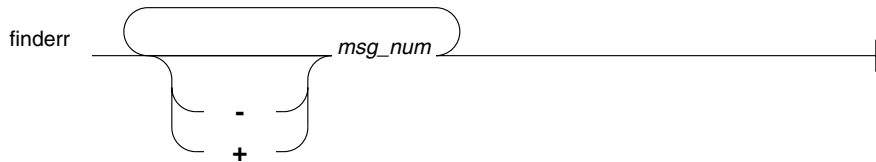
`$INFORMIXDIR/msg/errmsg.txt`

The error message numbers range from -1 to -33000. When you specify these numbers for the **finderr** or **rofferr** scripts, you can omit the minus sign. A few messages have positive numbers. In the unlikely event that you want to display them, you must precede the message number with a + sign.

The messages numbered -1 to -100 can be platform-dependent. If the message text for a message in this range does not apply to your platform, check the operating system documentation for the precise meaning of the message number.

The finderr Script

Use the **finderr** script to display one or more error messages, and their corrective actions, on the terminal screen. The **finderr** script has the following syntax:



You can specify any number of error messages per **finderr** command. The **finderr** command copies all the specified messages, and their corrective actions, to standard output.

For example, to display the -359 error message, you can enter the following command:

```
finderr -359
```

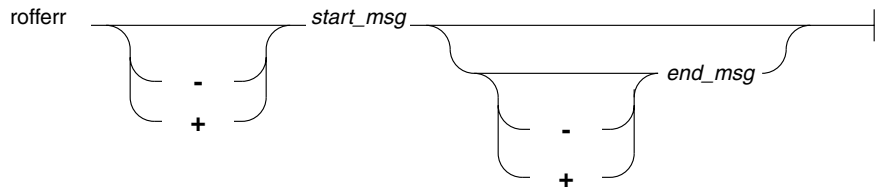
The following example demonstrates how to specify a list of error messages. This example also pipes the output to the UNIX **more** command to control the display. You can also redirect the output to another file so that you can save or print the error messages:

```
finderr 233 107 113 134 143 144 154 | more
```

The rofferr Script

Use the **rofferr** script to format one error message, or a range of error messages, for printing. By default, **rofferr** displays output on the screen. You need to send the output to **nroff** to interpret the formatting commands and then to a printer, or to a file where the **nroff** output is stored until you are ready to print. You can then print the file. For information on using **nroff** and on printing files, see your UNIX documentation.

The **rofferr** script has the following syntax:



The following example formats error message -359. It pipes the formatted error message into **nroff** and sends the output of **nroff** to the default printer:

```
rofferr 359 | nroff -man | lpr
```

The following example formats and then prints all the error messages between -1300 and -4999:

```
rofferr -1300 -4999 | nroff -man | lpr
```

Documentation Notes, Release Notes, Machine Notes

In addition to the IBM Informix set of manuals, the following on-line files, located in the `$INFORMIXDIR/release` directory, may supplement the information in *IBM Informix Guide to SQL: Reference*.

Online File	Purpose
<code>SQLRDOC_5.txt</code>	The documentation notes file describes features that are not covered in the manual or that were modified since publication.
<code>ENGREL_5.txt</code>	The release notes file describes feature differences from earlier versions of IBM Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
<code>ONLINE_5.txt</code>	The machine notes file describes any special actions that you must take to configure and use IBM Informix products on your computer. Machine notes are named for the product described.

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

A number of IBM Informix products also provide on-line Help files that walk you through each menu option. To invoke the Help feature, simply press CTRL-W wherever you are in your IBM Informix product.

Compliance with Industry Standards

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

IBM Welcomes Your Comments

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

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

Send electronic mail to us at the following address:

`docinf@us.ibm.com`

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

The stores5 Database

In This Chapter	1-3
Structure of the Tables	1-4
The customer Table	1-5
The orders Table	1-6
The items Table	1-6
The stock Table	1-8
The catalog Table	1-9
The cust_calls Table	1-10
The call_type Table	1-10
The manufact Table	1-11
The state Table	1-11
The stores5 Database Map	1-11
Primary-Foreign Key Relationships	1-13
The customer and orders Tables	1-14
The orders and items Tables	1-15
The items and stock Tables	1-16
The stock and catalog Tables	1-17
The stock and manufact Tables	1-18
The cust_calls and customer Tables	1-19
The call_type and cust_calls Table	1-20
The state and customer Tables	1-21
Data in the stores5 Database.	1-21

In This Chapter

The **stores5** database contains a set of tables that describe an imaginary business. The examples in this book are based on this database. The **stores5** database is not ANSI-compliant. Information on creating the **stores5** database appears in the Introduction of this manual.

This chapter contains four sections:

- The first section describes the structure of the tables in the **stores5** database. It identifies the primary key of each table, lists the name and data type of each column, and indicates whether the column has a default value or check constraint. Indexes on columns also are identified and classified as unique or as allowing duplicate values.
- The second section shows a graphic map of the tables in the **stores5** database and indicates the relationships between columns.
- The third section describes the primary-foreign key relationships between columns in tables.
- The final section shows the data contained in each table of the **stores5** database.

Structure of the Tables

The **stores5** database contains information about a fictitious sporting-goods distributor that services stores in the western United States. This database includes the following tables:

- **customer**
- **orders**
- **items**
- **stock**
- **catalog**
- **cust_calls**
- **call_type**
- **manufact**
- **state**

The following sections describe each table. The unique identifier for each table (primary key) is shaded and indicated by a key symbol.

The customer Table


The **customer** table contains information about the retail stores that place orders from the distributor. The columns of the **customer** table are as follows:

Column Name	Data Type	Description
customer_num	SERIAL(101)	system-generated customer number
fname	CHAR(15)	first name of store representative
lname	CHAR(15)	last name of store representative
company	CHAR(20)	name of store
address1	CHAR(20)	first line of store address
address2	CHAR(20)	second line of store address
city	CHAR(15)	city
state	CHAR(2)	state (foreign key to state table)
zipcode	CHAR(5)	zip code
phone	CHAR(18)	telephone number

The **zipcode** column is indexed to allow duplicate values.


The orders Table

The **orders** table contains information about orders placed by the customers of the distributor. The columns of the **orders** table are as follows:

Column Name	Data Type	Description
 order_num	SERIAL(1001)	system-generated order number
order_date	DATE	date order entered
customer_num	INTEGER	customer number (foreign key to customer table)
ship_instruct	CHAR(40)	special shipping instructions
backlog	CHAR(1)	indicates order cannot be filled because the item is backlogged: y = yes n = no
po_num	CHAR(10)	customer purchase order number
ship_date	DATE	shipping date
ship_weight	DECIMAL(8,2)	shipping weight
ship_charge	MONEY(6)	shipping charge
paid_date	DATE	date order paid

The items Table


An order can include one or more items. There is one row in the **items** table for each item in an order. The columns of the **items** table are as follows:

	Column Name	Data Type	Description
	item_num	SMALLINT	sequentially assigned item number for an order
	order_num	INTEGER	order number (foreign key to orders table)
	stock_num	SMALLINT	stock number for item (foreign key to stock table)
	manu_code	CHAR(3)	manufacturer code for item ordered (foreign key to manufact table)
	quantity	SMALLINT	quantity ordered (value must be > 1)
	total_price	MONEY(8)	quantity ordered × unit price = total price of item

The stock Table


The distributor carries 41 different types of sporting goods from various manufacturers. More than one manufacturer can supply a sporting good. For example, the distributor offers racer goggles from two manufacturers and running shoes from six manufacturers.

The **stock** table is a catalog of the items sold by the distributor. The columns of the **stock** table are as follows:

Column Name	Data Type	Description
 stock_num	SMALLINT	stock number that identifies type of item
manu_code	CHAR(3)	manufacturer code (foreign key to manufact table)
description	CHAR(15)	description of item
unit_price	MONEY(6,2)	unit price
unit	CHAR(4)	unit by which item is ordered: each pair case box
unit_descr	CHAR(15)	description of unit

The catalog Table


The **catalog** table describes each of the items in stock. Retail stores use this table when placing orders with the distributor. The columns of the **catalog** table are as follows:

Column Name	Data Type	Description
 catalog_num	SERIAL(10001)	system-generated catalog number
stock_num	SMALLINT	distributor stock number (foreign key to stock table)
manu_code	CHAR(3)	manufacturer code (foreign key to manufact table)
cat_descr	TEXT	description of item
cat_picture	BYTE	picture of item (binary data)
cat_advert	VARCHAR(255, 65)	tag line underneath picture

The **catalog** table appears only if you are using an IBM Informix OnLine database server.


The cust_calls Table

All customer calls for information on orders, shipments, or complaints are logged. The **cust_calls** table contains information about these types of customer calls. The columns of the **cust_calls** table are as follows:

	Column Name	Data Type	Description
	customer_num	INTEGER	customer number (foreign key to customer table)
	call_dtime	DATETIME YEAR TO MINUTE	date and time call received
	user_id	CHAR(18)	name of person logging call (default is user login name)
	call_code	CHAR(1)	type of call (foreign key to call_type table)
	call_descr	CHAR(240)	description of call
	res_dtime	DATETIME YEAR TO MINUTE	date and time call resolved
	res_descr	CHAR(240)	description of how call was resolved


The call_type Table

The call codes associated with customer calls are stored in the **call_type** table. The columns of the **call_type** table are as follows:

	Column Name	Data Type	Description
	call_code	CHAR(1)	call code
	code_descr	CHAR (30)	description of call type


The manufact Table

Information about the nine manufacturers whose sporting goods are handled by the distributor is stored in the **manufact** table. The columns of the **manufact** table are as follows:

Column Name	Data Type	Description
 manu_code	CHAR(3)	manufacturer code
manu_name	CHAR(15)	name of manufacturer
lead_time	INTERVAL DAY(3) TO DAY	lead time for shipment of orders

The state Table

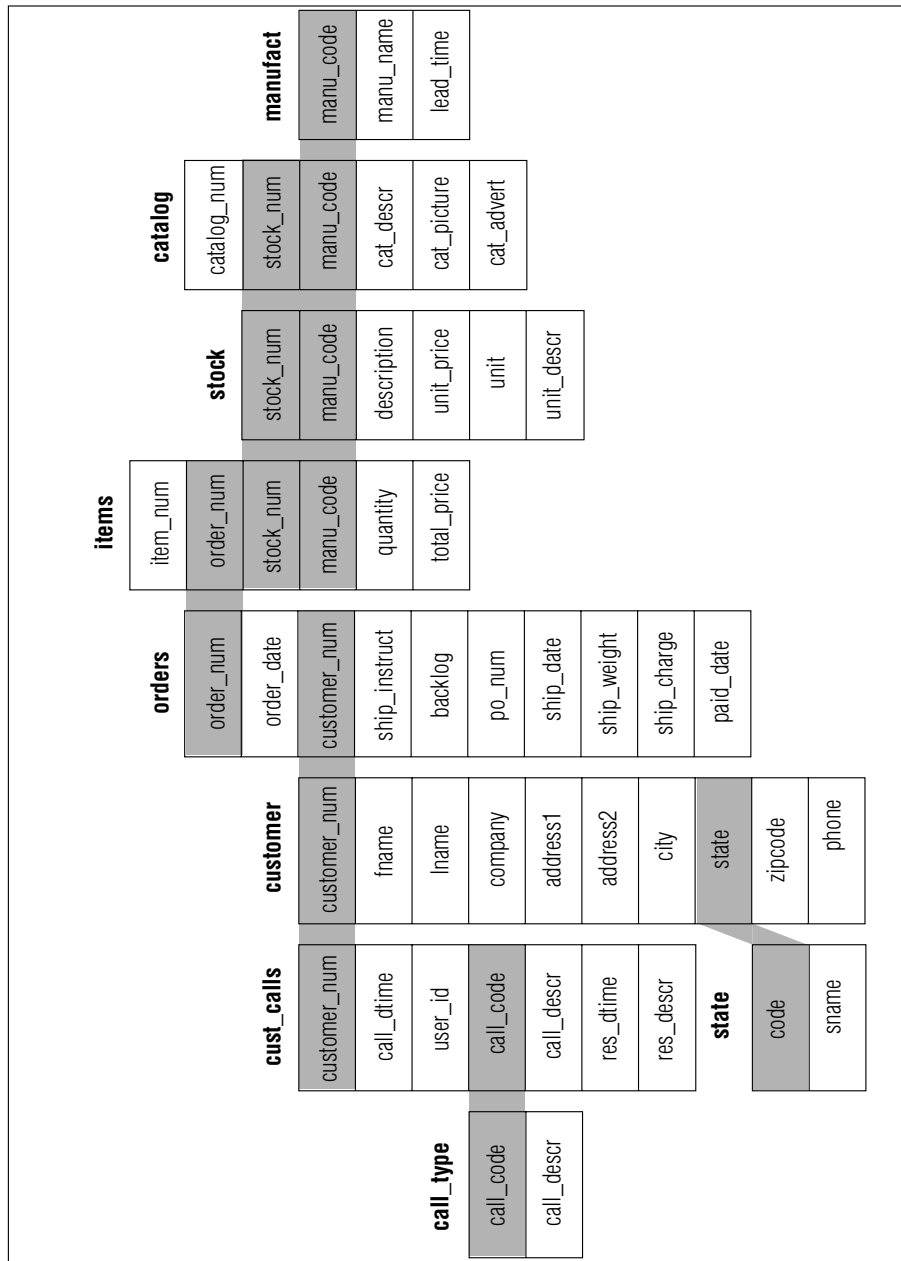
The **state** table contains the names and postal abbreviations for the 50 states of the United States. The columns of the **state** table are as follows:

Column Name	Data Type	Description
 code	CHAR(2)	state code
sname	CHAR(15)	state name

The stores5 Database Map

Figure 1-1 displays the joins in the **stores5** database. The lines connecting a column in one table to the same column in another table indicate the relationships, or joins, *between* tables.

Figure 1-1
Joins in the stores5 database



Primary-Foreign Key Relationships

The tables of the **stores5** database are linked by the primary-foreign key relationships shown in [Figure 1-1](#) and identified in this section. This type of relationship is called a *referential constraint* because a foreign key in one table *references* the primary key in another table. [Figure 1-2](#) through [Figure 1-8](#) show the relationships among tables and how information stored in one table supplements information stored in others.

The customer and orders Tables

The **customer** table contains a **customer_num** column that holds a number identifying a customer, along with columns for the customer name, company, address, and telephone number. For example, the row with information about Anthony Higgins contains the number 104 in the **customer_num** column. The **orders** table also contains a **customer_num** column that stores the number of the customer who placed a particular order. In the **orders** table, the **customer_num** column is a foreign key that references the **customer_num** column in the **customer** table. This relationship is shown in [Figure 1-2](#).

Figure 1-2
Tables joined by the **customer_num** column

customer Table (detail)		
customer_num	fname	lname
101	Ludwig	Pauli
102	Carole	Sadler
103	Philip	Currie
104	Anthony	Higgins

orders Table (detail)		
order_num	order_date	customer_num
1001	05/20/1991	104
1002	05/21/1991	101
1003	05/22/1991	104
1004	05/22/1991	106

According to [Figure 1-2](#), customer 104 (Anthony Higgins) has placed two orders, since his customer number appears in two rows of the **orders** table. Since the customer number is a foreign key in the **orders** table, you can retrieve Anthony Higgins' name, address, and information about his orders at the same time.

The orders and items Tables

The **orders** and **items** tables are linked by an **order_num** column that contains an identification number for each order. If an order includes several items, the same order number appears in several rows of the **items** table. In the **items** table, the **order_num** column is a foreign key that references the **order_num** column in the **orders** table. [Figure 1-3](#) shows this relationship.

Figure 1-3
Tables joined by the **order_num** column

orders Table (detail)		
order_num	order_date	customer_num
1001	05/20/1991	104
1002	05/21/1991	101
1003	05/22/1991	104

items Table (detail)			
item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ

The items and stock Tables

The **items** table and the **stock** table are joined by two columns: the **stock_num** column stores a stock number for an item, and the **manu_code** column stores a code that identifies the manufacturer. You need both the stock number and the manufacturer code to uniquely identify an item. For example, the item with the stock number 1 and the manufacturer code HRO is a Hero baseball glove, while the item with the stock number 1 and the manufacturer code HSK is a Husky baseball glove. The same stock number and manufacturer code can appear in more than one row of the **items** table, if the same item belongs to separate orders. In the **items** table, the **stock_num** and **manu_code** columns are foreign keys that reference the **stock_num** and **manu_code** columns in the **stock** table. This is illustrated in [Figure 1-4](#).

Figure 1-4

Tables joined by the **stock_num** and **manu_code** columns

items Table (detail)				
item_num	order_num	stock_num	manu_code	
1	1001	1	HRO	
1	1002	4	HSK	
2	1002	3	HSK	
1	1003	9	ANZ	
2	1003	8	ANZ	
3	1003	5	ANZ	
1	1004	1	HRO	

stock Table (detail)		
stock_num	manu_code	description
1	HRO	baseball gloves
1	HSK	baseball gloves
1	SMT	baseball gloves

The stock and catalog Tables

The **stock** table and **catalog** table are joined by two columns: the **stock_num** column, which stores a stock number for an item, and the **manu_code** column, which stores a code that identifies the manufacturer. You need both columns to uniquely identify an item. In the **catalog** table, the **stock_num** and **manu_code** columns are foreign keys that reference the **stock_num** and **manu_code** columns in the **stock** table. [Figure 1-5](#) shows this relationship.

Figure 1-5

Tables joined by the **stock_num** and **manu_code** columns

stock Table (detail)		
stock_num	manu_code	description
1	HRO	baseball gloves
1	HSK	baseball gloves
1	SMT	baseball gloves

catalog Table (detail)		
catalog_num	stock_num	manu_code
10001	1	HRO
10002	1	HSK
10003	1	SMT
10004	2	HRO

The stock and manufact Tables

The **stock** table and the **manufact** table are joined by the **manu_code** column. The same manufacturer code can appear in more than one row of the **stock** table if the manufacturer produces more than one piece of equipment. In the **stock** table, the **manu_code** column is a foreign key that references the **manu_code** column in the **manufact** table. This relationship is illustrated in [Figure 1-6](#).

Figure 1-6
Tables joined by the **manu_code** column

stock Table (detail)		
stock_num	manu_code	description
1	HRO	baseball gloves
1	HSK	baseball gloves
1	SMT	baseball gloves

manufact Table (detail)	
manu_code	manu_name
NRG	Norge
HSK	Husky
HRO	Hero

The cust_calls and customer Tables

The **cust_calls** table and the **customer** table are joined by the **customer_num** column. The same customer number can appear in more than one row of the **cust_calls** table if the customer calls the distributor more than once with a problem or question. In the **cust_calls** table, the **customer_num** column is a foreign key that references the **customer_num** column in the **customer** table. This relationship is illustrated in [Figure 1-7](#).

Figure 1-7
Tables joined by the **customer_num** column

customer Table (detail)		
customer_num	fname	lname
101	Ludwig	Pauli
102	Carole	Sadler
103	Philip	Currie
104	Anthony	Higgins
105	Raymond	Vector
106	George	Watson

cust_calls Table (detail)		
customer_num	call_dtime	user_id
106	1991-06-12 08:20	maryj
127	1991-07-31 14:30	maryj
116	1990-11-28 13:34	mannyh
116	1990-12-21 11:24	mannyh

The call_type and cust_calls Table

The `call_type` and `cust_calls` tables are joined by the `call_code` column. The same call code can appear in more than one row of the `cust_calls` table since many customers can have the same *type* of problem. In the `cust_calls` table, the `call_code` column is a foreign key that references the `call_code` column in the `call_type` table. This relationship is illustrated in [Figure 1-8](#).

Figure 1-8
Tables joined by the `call_code` column

call_type Table (detail)	
call_code	code_descr
B	billing error
D	damaged goods
I	incorrect merchandise sent
L	late shipment
O	other

cust_calls Table (detail)		
customer_num	call_dtime	call_code
106	1991-06-12 08:20	D
127	1991-07-31 14:30	I
116	1990-11-28 13:34	I
116	1990-12-21 11:24	I

The state and customer Tables

The **state** table and the **customer** table are joined by a column that contains the state code. This column is called **code** in the **state** table and **state** in the **customer** table. If several customers live in the same state, the same state code appears in several rows of the table. In the **customer** table, the **state** column is a foreign key that references the **code** column in the **state** table. This is shown in [Figure 1-9](#).

Figure 1-9
Tables joined by the **state/code** column

customer Table (detail)				
customer_num	fname	lname	---	state
101	Ludwig	Pauli	---	CA
102	Carole	Sadler	---	CA
103	Philip	Currie	---	CA

state Table (detail)	
code	sname
AK	Alaska
AL	Alabama
AR	Arkansas
AZ	Arizona
CA	California

Data in the stores5 Database

The tables that follow display the data in the **stores5** database.

customer Table

customer_num	fname	lname	company	address1	address2	city	state	zipcode	phone
101	Ludwig	Pauli	All Sports Supplies	213 Erstwild Court		Sunnyvale	CA	94086	408-789-8075
102	Carole	Sadler	Sports Spot	785 Geary St		San Francisco	CA	94117	415-822-1289
103	Philip	Currie	Phil's Sports	654 Poplar	P. O. Box 3498	Palo Alto	CA	94303	415-328-4543
104	Anthony	Higgins	Play Ball!	East Shopping Cntr.	422 Bay Road	Redwood City	CA	94026	415-368-1100
105	Raymond	Vector	Los Altos Sports	1899 La Loma Drive		Los Altos	CA	94022	415-776-3249
106	George	Watson	Watson & Son	1143 Carver Place		Mountain View	CA	94063	415-389-8789
107	Charles	Ream	Athletic Supplies	41 Jordan Avenue		Palo Alto	CA	94304	415-356-9876
108	Donald	Quinn	Quinn's Sports	587 Alvarado		Redwood City	CA	94063	415-544-8729
109	Jane	Miller	Sport Stuff	Mayfair Mart	7345 Ross Blvd.	Sunnyvale	CA	94086	408-723-8789
110	Roy	Jaeger	AA Athletics	520 Topaz Way		Redwood City	CA	94062	415-743-3611
111	Frances	Keyes	Sports Center	3199 Sterling Court		Sunnyvale	CA	94085	408-277-7245
112	Margaret	Lawson	Runners & Others	234 Wyandotte Way		Los Altos	CA	94022	415-887-7235
113	Lana	Beatty	Sportstown	654 Oak Grove		Sunnyvale	CA	94085	408-277-7245
114	Frank	Albertson	Sporting Place	947 Waverly Place		Menlo Park	CA	94025	415-356-9982
115	Alfred	Grant	Gold Medal Sports	776 Gary Avenue		Redwood City	CA	94062	415-886-6677
116	Jean	Parmelee	Olympic City	1104 Spinosa Drive		Menlo Park	CA	94025	415-356-1123
117	Arnold	Sipes	Kids Korner	850 Lytton Court		Mountain View	CA	94040	415-534-8822
118	Dick	Baxter	Blue Ribbon Sports	5427 College		Redwood City	CA	94063	415-245-4578
119	Bob	Shorter	The Triathletes Club	2405 Kings Highway		Oakland	CA	94609	415-655-0011
120	Fred	Jewell	Century Pro Shop	6627 N. 17th Way	350 W. 23rd Street	Cherry Hill	NJ	08002	609-663-6079
121	Jason	Wallack	City Sports	Lake Biltmore Mall		Phoenix	AZ	85016	602-265-8754
122	Cathy	O'Brian	The Sporting Life	543 Nassau Street		Wilmington	DE	19898	302-366-7511
123	Marvin	Hanlon	Bay Sports	10100 Bay Meadows Rd	Suite 1020	Princeton	NJ	08540	609-342-0054
124	Chris	Putnum	Putnum's Putters	4715 S.E. Adams Blvd	Suite 909C	Jacksonville	FL	32256	904-823-4239
125	James	Henry	Total Fitness Sports	1450 Commonwealth Av		Bartlesville	OK	74006	918-355-2074
126	Eileen	Neele	Neele's Discount Sp	2539 South Utica St		Brighton	MA	02135	617-232-4159
127	Kim	Satifer	Big Blue Bike Shop	Blue Island Square	12222 Gregory Street	Denver	CO	80219	303-936-7731
128	Frank	Lessor	Phoenix University	Athletic Department	1817 N. Thomas Road	Blue Island	NY	60406	312-944-5691
						Phoenix	AZ	85008	602-533-1817

items Table (1 of 2)

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1001	1	HRO	1	250.00
1	1002	4	HSK	1	960.00
2	1002	3	HSK	1	240.00
1	1003	9	ANZ	1	20.00
2	1003	8	ANZ	1	840.00
3	1003	5	ANZ	5	99.00
1	1004	1	HRO	1	250.00
2	1004	2	HRO	1	126.00
3	1004	3	HSK	1	240.00
4	1004	1	HSK	1	800.00
1	1005	5	NRG	10	280.00
2	1005	5	ANZ	10	198.00
3	1005	6	SMT	1	36.00
4	1005	6	ANZ	1	48.00
1	1006	5	SMT	5	125.00
2	1006	5	NRG	5	140.00
3	1006	5	ANZ	5	99.00
4	1006	6	SMT	1	36.00
5	1006	6	ANZ	1	48.00
1	1007	1	HRO	1	250.00
2	1007	2	HRO	1	126.00
3	1007	3	HSK	1	240.00
4	1007	4	HRO	1	480.00
5	1007	7	HRO	1	600.00
1	1008	8	ANZ	1	840.00
2	1008	9	ANZ	5	100.00
1	1009	1	SMT	1	450.00
1	1010	6	SMT	1	36.00
2	1010	6	ANZ	1	48.00
1	1011	5	ANZ	5	99.00
1	1012	8	ANZ	1	840.00
2	1012	9	ANZ	10	200.00
1	1013	5	ANZ	1	19.80
2	1013	6	SMT	1	36.00
3	1013	6	ANZ	1	48.00
4	1013	9	ANZ	2	40.00
1	1014	4	HSK	1	960.00
2	1014	4	HRO	1	480.00
1	1015	1	SMT	1	450.00
1	1016	101	SHM	2	136.00
2	1016	109	PRC	3	90.00

items Table (2 of 2)

item_num	order_num	stock_num	manu_code	quantity	total_price
3	1016	110	HSK	1	308.00
4	1016	114	PRC	1	120.00
1	1017	201	NKL	4	150.00
2	1017	202	KAR	1	230.00
3	1017	301	SHM	2	204.00
1	1018	307	PRC	2	500.00
2	1018	302	KAR	3	15.00
3	1018	110	PRC	1	236.00
4	1018	5	SMT	4	100.00
5	1018	304	HRO	1	280.00
1	1019	111	SHM	3	1499.97
1	1020	204	KAR	2	90.00
2	1020	301	KAR	4	348.00
1	1021	201	NKL	2	75.00
2	1021	201	ANZ	3	225.00
3	1021	202	KAR	3	690.00
4	1021	205	ANZ	2	624.00
1	1022	309	HRO	1	40.00
2	1022	303	PRC	2	96.00
3	1022	6	ANZ	2	96.00
1	1023	103	PRC	2	40.00
2	1023	104	PRC	2	116.00
3	1023	105	SHM	1	80.00
4	1023	110	SHM	1	228.00
5	1023	304	ANZ	1	170.00
6	1023	306	SHM	1	190.00

call_type Table

call_code	code_descr
B	billing error
D	damaged goods
I	incorrect merchandise sent
L	late shipment
O	other

orders Table

order_num	order_date	customer_num	ship_instruct	backlog	po_num	ship_date	ship_weight	ship_charge	paid_date
1001	05/20/1991	104	express	n	B77836	06/01/1991	20.40	10.00	07/22/1991
1002	05/21/1991	101	PO on box, deliver back door only	n	9270	05/26/1991	15.30	15.30	06/03/1991
1003	05/22/1991	104	express	n	B77890	05/23/1991	50.60	10.80	06/14/1991
1004	05/22/1991	106	ring bell twice	y	8006	05/30/1991	35.60	10.80	06/14/1991
1005	05/24/1991	116	call before delivery	n	2865	05/30/1991	95.80	19.20	06/21/1991
1006	05/30/1991	112	after 10 am	y	Q13557	06/09/1991	80.80	16.20	06/21/1991
1007	05/31/1991	117		y	278693	06/05/1991	70.80	14.20	
1008	06/07/1991	110	closed Monday	n	278693	06/05/1991	125.90	25.20	
1009	06/14/1991	111	door next to grocery	y	LZ230	07/06/1991	45.60	13.80	07/21/1991
1010	06/17/1991	115	deliver 776 King St. if no answer	n	4745	06/21/1991	20.40	10.00	08/21/1991
1011	06/18/1991	104	express	n	429Q	06/29/1991	40.60	12.30	08/22/1991
1012	06/18/1991	117		n	B77897	07/03/1991	10.40	5.00	08/29/1991
1013	06/22/1991	104	express	n	278701	06/29/1991	70.80	14.20	
1014	06/25/1991	106	ring bell, kick door loudly	n	B77930	07/10/1991	60.80	12.20	07/31/1991
1015	06/27/1991	110	closed Mondays	n	8052	07/03/1991	40.60	12.30	07/10/1991
1016	06/29/1991	119	delivery entrance off Camp St.	n	MA003	07/16/1991	20.60	6.30	08/31/1991
1017	07/09/1991	120	North side of clubhouse	n	PC6782	07/12/1991	35.00	11.80	
1018	07/10/1991	121	SW corner of Biltmore Mall	n	DM354331	07/13/1991	60.00	18.00	08/06/1991
1019	07/11/1991	122	closed til noon Mondays	n	S22942	07/13/1991	70.50	20.00	08/06/1991
1020	07/11/1991	123	express	n	Z55709	07/16/1991	90.00	23.00	09/20/1991
1021	07/23/1991	124	ask for Elaine	n	W2286	07/16/1991	14.00	8.50	09/20/1991
1022	07/24/1991	126	express	n	C3288	07/25/1991	40.00	12.00	08/22/1991
1023	07/24/1991	127	no deliveries after 3 p.m.	n	W9925	07/30/1991	15.00	13.00	09/02/1991
				n	KF2961	07/30/1991	60.00	18.00	08/22/1991

stock Table (1 of 2)

stock_num	manu_code	description	unit_price	unit	unit_descr
1	HRO	baseball gloves	250.00	case	10 gloves/case
1	HSK	baseball gloves	800.00	case	10 gloves/case
1	SMT	baseball gloves	450.00	case	10 gloves/case
2	HRO	baseball	126.00	case	24/case
3	HSK	baseball bat	240.00	case	12/case
4	HSK	football	960.00	case	24/case
4	HRO	football	480.00	case	24/case
5	NRG	tennis racquet	28.00	each	each
5	SMT	tennis racquet	25.00	each	each
5	ANZ	tennis racquet	19.80	each	each
6	SMT	tennis ball	36.00	case	24 cans/case
6	ANZ	tennis ball	48.00	case	24 cans/case
7	HRO	basketball	600.00	case	24/case
8	ANZ	volleyball	840.00	case	24/case
9	ANZ	volleyball net	20.00	each	each
101	PRC	bicycle tires	88.00	box	4/box
101	SHM	bicycle tires	68.00	box	4/box
102	SHM	bicycle brakes	220.00	case	4 sets/case
102	PRC	bicycle brakes	480.00	case	4 sets/case
103	PRC	front derailleur	20.00	each	each
104	PRC	rear derailleur	58.00	each	each
105	PRC	bicycle wheels	53.00	pair	pair
105	SHM	bicycle wheels	80.00	pair	pair
106	PRC	bicycle stem	23.00	each	each
107	PRC	bicycle saddle	70.00	pair	pair
108	SHM	crankset	45.00	each	each
109	PRC	pedal binding	30.00	case	6 pairs/case
109	SHM	pedal binding	200.00	case	4 pairs/case
110	PRC	helmet	236.00	case	4/case
110	ANZ	helmet	244.00	case	4/case
110	SHM	helmet	228.00	case	4/case
110	HRO	helmet	260.00	case	4/case
110	HSK	helmet	308.00	case	4/case
111	SHM	10-spdl, assmbld	499.99	each	each
112	SHM	12-spdl, assmbld	549.00	each	each
113	SHM	18-spdl, assmbld	685.90	each	each
114	PRC	bicycle gloves	120.00	case	10 pairs/case

stock Table (2 of 2)

stock_num	manu_code	description	unit_price	unit	unit_descr
201	NKL	golf shoes	37.50	each	each
201	ANZ	golf shoes	75.00	each	each
201	KAR	golf shoes	90.00	each	each
202	NKL	metal woods	174.00	case	2 sets/case
202	KAR	std woods	230.00	case	2 sets/case
203	NKL	irons/wedges	670.00	case	2 sets/case
204	KAR	putter	45.00	each	each
205	NKL	3 golf balls	312.00	case	24/case
205	ANZ	3 golf balls	312.00	case	24/case
205	HRO	3 golf balls	312.00	case	24/case
301	NKL	running shoes	97.00	each	each
301	HRO	running shoes	42.50	each	each
301	SHM	running shoes	102.00	each	each
301	PRC	running shoes	75.00	each	each
301	KAR	running shoes	87.00	each	each
301	ANZ	running shoes	95.00	each	each
302	HRO	ice pack	4.50	each	each
302	KAR	ice pack	5.00	each	each
303	PRC	socks	48.00	box	24 pairs/box
303	KAR	socks	36.00	box	24 pair/box
304	ANZ	watch	170.00	box	10/box
304	HRO	watch	280.00	box	10/box
305	HRO	first-aid kit	48.00	case	4/case
306	PRC	tandem adapter	160.00	each	each
306	SHM	tandem adapter	190.00	each	each
307	PRC	infant jogger	250.00	each	each
308	PRC	twin jogger	280.00	each	each
309	HRO	ear drops	40.00	case	20/case
309	SHM	ear drops	40.00	case	20/case
310	SHM	kick board	80.00	case	10/case
310	ANZ	kick board	89.00	case	12/case
311	SHM	water gloves	48.00	box	4 pairs/box
312	SHM	racer goggles	96.00	box	12/box
312	HRO	racer goggles	72.00	box	12/box
313	SHM	swim cap	72.00	box	12/box
313	ANZ	swim cap	60.00	box	12/box

catalog Table (1 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10001	1	HRO	Brown leather. Specify first baseman's or infield/outfield style. Specify right- or left-handed.	<BYTE value>	Your First Season's Baseball Glove
10002	1	HSK	Babe Ruth signature glove. Black leather. Infield/outfield style. Specify right- or left-handed.	<BYTE value>	All-Leather, Hand-Stitched, Deep-Pockets, Sturdy Webbing that Won't Let Go
10003	1	SMT	Catcher's mitt. Brown leather. Specify right- or left-handed.	<BYTE value>	A Sturdy Catcher's Mitt With the Perfect Pocket
10004	2	HRO	Jackie Robinson signature glove. Highest Professional quality, used by National League.	<BYTE value>	Highest Quality Ball Available, from the Hand-Stitching to the Robinson Signature
10005	3	HSK	Pro-style wood. Available in sizes: 31, 32, 33, 34, 35.	<BYTE value>	High-Technology Design Expands the Sweet Spot
10006	3	SHM	Aluminum. Blue with black tape. 31", 20 oz or 22 oz; 32", 21 oz or 23 oz; 33", 22 oz or 24 oz.	<BYTE value>	Durable Aluminum for High School and Collegiate Athletes
10007	4	HSK	Norm Van Brocklin signature style.	<BYTE value>	Quality Pigskin with Norm Van Brocklin Signature
10008	4	HRO	NFL-Style pigskin.	<BYTE value>	Highest Quality Football for High School and Collegiate Competitions
10009	5	NRG	Graphite frame. Synthetic strings.	<BYTE value>	Wide Body Amplifies Your Natural Abilities by Providing More Power Through Aerodynamic Design
10010	5	SMT	Aluminum frame. Synthetic strings.	<BYTE value>	Mid-Sized Racquet For the Improving Player
10011	5	ANZ	Wood frame, cat-gut strings.	<BYTE value>	Antique Replica of Classic Wooden Racquet Built with Cat-Gut Strings
10012	6	SMT	Soft yellow color for easy visibility in sunlight or artificial light.	<BYTE value>	High-Visibility Tennis, Day or Night
10013	6	ANZ	Pro-core. Available in neon yellow, green, and pink.	<BYTE value>	Durable Construction Coupled with the Brightest Colors Available
10014	7	HRO	Indoor. Classic NBA style. Brown leather.	<BYTE value>	Long-Life Basketballs for Indoor Gymnasiums
10015	8	ANZ	Indoor. Finest leather. Professional quality.	<BYTE value>	Professional Volleyballs for Indoor Competitions
10016	9	ANZ	Steel eyelets. Nylon cording. Double-stitched. Sanctioned by the National Athletic Congress.	<BYTE value>	Sanctioned Volleyball Netting for Indoor Professional and Collegiate Competition

catalog Table (2 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10017	101	PRC	Reinforced, hand-finished tubular. Polyurethane belted. Effective against punctures. Mixed tread for super wear and road grip.	<BYTE value>	Ultimate in Puncture Protection, Tires Designed for In-City Riding
10018	101	SHM	Durable nylon casing with butyl tube for superior air retention. Center-ribbed tread with herringbone side. Coated sidewalls resist abrasion.	<BYTE value>	The Perfect Tire for Club Rides or Training
10019	102	SHM	Thrust bearing and coated pivot washer/spring sleeve for smooth action. Slotted levers with soft gum hoods. Two-tone paint treatment. Set includes calipers, levers, and cables.	<BYTE value>	Thrust-Bearing and Spring-Sleeve Brake Set Guarantees Smooth Action
10020	102	PRC	Computer-aided design with low-profile pads. Cold-forged alloy calipers and beefy caliper bushing. Aero levers. Set includes calipers, levers, and cables.	<BYTE value>	Computer Design Delivers Rigid Yet Vibration-Free Brakes
10021	103	PRC	Compact leading-action design enhances shifting. Deep cage for super-small granny gears. Extra strong construction to resist off-road abuse.	<BYTE value>	Climb Any Mountain: ProCycle's Front Deraillleur Adds Finesse to Your ATB
10022	104	PRC	Floating trapezoid geometry with extra thick parallelogram arms. 100-tooth capacity. Optimum alignment with any freewheel.	<BYTE value>	Computer-Aided Design Engineers 100-Tooth Capacity Into ProCycle's Rear Deraillleur
10023	105	PRC	Front wheels laced with 15g spokes in a 3-cross pattern. Rear wheels laced with 14g spikes in a 3-cross pattern.	<BYTE value>	Durable Training Wheels That Hold True Under Toughest Conditions
10024	105	SHM	Polished alloy. Sealed-bearing, quick-release hubs. Double-butted. Front wheels are laced 15g/2-cross. Rear wheels are laced 15g/3-cross.	<BYTE value>	Extra Lightweight Wheels for Training or High-Performance Touring
10025	106	PRC	Hard anodized alloy with pearl finish. 6mm hex bolt hardware. Available in lengths of 90-140mm in 10mm increments.	<BYTE value>	ProCycle Stem with Pearl Finish
10026	107	PRC	Available in three styles: Mens racing; Mens touring; and Women's. Anatomical gel construction with lycra cover. Black or black/hot pink.	<BYTE value>	The Ultimate In Riding Comfort, Lightweight With Anatomical Support

catalog Table (3 of 7)

catalog_num	stock_num	manu_code	cat_desc	cat_picture	cat_advert
10027	108	SHM	Double or triple crankset with choice of chainrings. For double crankset, chainrings from 38-54 teeth. For triple crankset, chainrings from 24-48 teeth.	<BYTE value>	Customize Your Mountain Bike With Extra-Durable Crankset
10028	109	PRC	Steel toe clips with nylon strap. Extra wide at buckle to reduce pressure.	<BYTE value>	Classic Toeclip Improved To Prevent Soreness At Clip Buckle
10029	109	SHM	Ingenious new design combines button on sole of shoe with slot on a pedal plate to give riders new options in riding efficiency. Choose full or partial locking. Four plates mean both top and bottom of pedals are slotted—no fishing around when you want to engage full power. Fast unlocking ensures safety when maneuverability is paramount.	<BYTE value>	Ingenious Pedal/Clip Design Delivers Maximum Power And Fast Unlocking
10030	110	PRC	Super-lightweight. Meets both ANZI and Snell standards for impact protection. 7.5 oz. Quick-release shadow buckle.	<BYTE value>	Feather-Light, Quick-Release, Maximum Protection Helmet
10031	110	ANZ	No buckle so no plastic touches your chin. Meets both ANZI and Snell standards for impact protection. 7.5 oz. Lycra cover.	<BYTE value>	Minimum Chin Contact, Feather-Light, Maximum Protection Helmet
10032	110	SHM	Dense outer layer combines with softer inner layer to eliminate the mesh cover, no snagging on brush. Meets both ANZI and Snell standards for impact protection. 8.0 oz.	<BYTE value>	Mountain Bike Helmet: Smooth Cover Eliminates the Worry of Brush Snags But Delivers Maximum Protection
10033	110	HRO	Newest ultralight helmet uses plastic shell. Largest ventilation channels of any helmet on the market. 8.5 oz.	<BYTE value>	Lightweight Plastic with Vents Assures Cool Comfort Without Sacrificing Protection
10034	110	HSK	Aerodynamic (teardrop) helmet covered with anti-drag fabric. Credited with shaving 2 seconds/mile from winner's time in Tour de France time-trial. 7.5 oz.	<BYTE value>	Teardrop Design Used by Yellow Jersey, You Can Time the Difference
10035	111	SHM	Light-action shifting 10 speed. Designed for the city commuter with shock-absorbing front fork and drilled eyelets for carry-all racks or bicycle trailers. Internal wiring for generator lights. 33 lbs.	<BYTE value>	Fully Equipped Bicycle Designed for the Serious Commuter Who Mixes Business With Pleasure

catalog Table (4 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10036	112	SHM	Created for the beginner enthusiast. Ideal for club rides and light touring. Sophisticated triple-butted frame construction. Precise index shifting. 28 lbs.	<BYTE value>	We Selected the Ideal Combination of Touring Bike Equipment, Then Turned It Into This Package Deal: High-Performance on the Roads, Maximum Pleasure Everywhere
10037	113	SHM	Ultra-lightweight. Racing frame geometry built for aerodynamic handlebars. Cantilever brakes. Index shifting. High-performance gearing. Quick-release hubs. Disk wheels. Bladed spokes.	<BYTE value>	Designed for the Serious Competitor, The Complete Racing Machine
10038	114	PRC	Padded leather palm and stretch mesh merged with terry back; Available in tan, black, and cream. Sizes S, M, L, XL.	<BYTE value>	Riding Gloves For Comfort and Protection
10039	201	NKL	Designed for comfort and stability. Available in white & blue or white & brown. Specify size.	<BYTE value>	Full-Comfort, Long-Wearing Golf Shoes for Men and Women
10040	201	ANZ	Guaranteed waterproof. Full leather upper. Available in white, bone, brown, green, and blue. Specify size.	<BYTE value>	Waterproof Protection Ensures Maximum Comfort and Durability In All Climates
10041	201	KAR	Leather and leather mesh for maximum ventilation. Waterproof lining to keep feet dry. Available in white & gray or white & ivory. Specify size.	<BYTE value>	Karsten's Top Quality Shoe Combines Leather and Leather Mesh
10042	202	NKL	Complete starter set utilizes gold shafts. Balanced for power.	<BYTE value>	Starter Set of Woods, Ideal for High School and Collegiate Classes
10043	202	KAR	Full set of woods designed for precision control and power performance.	<BYTE value>	High-Quality Woods Appropriate for High School Competitions or Serious Amateurs
10044	203	NKL	Set of eight irons includes 3 through 9 irons and pitching wedge. Originally priced at \$489.00.	<BYTE value>	Set of Irons Available From Factory at Tremendous Savings: Discontinued Line
10045	204	KAR	Ideally balanced for optimum control. Nylon-covered shaft.	<BYTE value>	High-Quality Beginning Set of Irons Appropriate for High School Competitions
10046	205	NKL	Fluorescent yellow.	<BYTE value>	Long Drive Golf Balls: Fluorescent Yellow
10047	205	ANZ	White only.	<BYTE value>	Long Drive Golf Balls: White
10048	205	HRO	Combination fluorescent yellow and standard white.	<BYTE value>	HiFlier Golf Balls: Case Includes Fluorescent Yellow and Standard White

catalog Table (5 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10049	301	NKL	Super shock-absorbing gel pads disperse vertical energy into a horizontal plane for extraordinary cushioned comfort. Great motion control. Mens only. Specify size.	<BYTE value>	Maximum Protection For High-Mileage Runners
10050	301	HRO	Engineered for serious training with exceptional stability. Fabulous shock absorption. Great durability. Specify mens/women's, size.	<BYTE value>	Pronators and Supinators Take Heart: A Serious Training Shoe For Runners Who Need Motion Control
10051	301	SHM	For runners who log heavy miles and need a durable, supportive, stable platform. Mesh/synthetic upper gives excellent moisture dissipation. Stability system uses rear antipronation platform and forefoot control plate for extended protection during high-intensity training. Specify mens/women's, size.	<BYTE value>	The Training Shoe Engineered for Marathoners and Ultra-Distance Runners
10052	301	PRC	Supportive, stable racing flat. Plenty of forefoot cushioning with added motion control. Women's only. D widths available. Specify size.	<BYTE value>	A Woman's Racing Flat That Combines Extra Forefoot Protection With a Slender Heel
10053	301	KAR	Anatomical last holds your foot firmly in place. Feather-weight cushioning delivers the responsiveness of a racing flat. Specify mens/women's, size.	<BYTE value>	Durable Training Flat That Can Carry You Through Marathon Miles
10054	301	ANZ	Cantilever sole provides shock absorption and energy rebound. Positive traction shoe with ample toe box. Ideal for runners who need a wide shoe. Available in mens and women's. Specify size.	<BYTE value>	Motion Control, Protection, and Extra Toebox Room
10055	302	KAR	Re-usable ice pack with velcro strap. For general use. Velcro strap allows easy application to arms or legs.	<BYTE value>	Finally, An Ice Pack for Achilles Injuries and Shin Splints that You Can Take to the Office
10056	303	PRC	Neon nylon. Perfect for running or aerobics. Indicate color: Fluorescent pink, yellow, green, and orange.	<BYTE value>	Knock Their Socks Off With YOUR Socks
10057	303	KAR	100% nylon blend for optimal wicking and comfort. We've taken out the cotton to eliminate the risk of blisters and reduce the opportunity for infection. Specify mens or women's.	<BYTE value>	100% Nylon Blend Socks - No Cotton

catalog Table (6 of 7)

catalog_num	stock_num	manu_code	cat_desc	cat_picture	cat_advert
10058	304	ANZ	Provides time, date, dual display of lap / cumulative splits, 4-lap memory, 10hr count-down timer, event timer, alarm, hour chime, waterproof to 50m, velcro band.	<BYTE value>	Athletic Watch w /4-Lap Memory
10059	304	HRO	Split timer, waterproof to 50m. Indicate color: Hot pink, mint green, space black.	<BYTE value>	Waterproof Triathlete Watch In Competition Colors
10060	305	HRO	Contains ace bandage, anti-bacterial cream, alcohol cleansing pads, adhesive bandages of assorted sizes, and instant-cold pack.	<BYTE value>	Comprehensive First-Aid Kit Essential for Team Practices, Team Traveling
10061	306	PRC	Converts a standard tandem bike into an adult/child bike. User-tested Assembly Instructions	<BYTE value>	Enjoy Bicycling With Your Child On a Tandem; Make Your Family Outing Safer
10062	306	SHIM	Converts a standard tandem bike into an adult/child bike. Lightweight model.	<BYTE value>	Consider a Touring Vacation For the Entire Family: A Lightweight, Touring Tandem for Parent and Child
10063	307	PRC	Allows mom or dad to take the baby out, too. Fits children up to 21 pounds. Navy blue with black trim.	<BYTE value>	Infant Jogger Keeps A Running Family Together
10064	308	PRC	Allows mom or dad to take both children! Rated for children up to 18 pounds.	<BYTE value>	As Your Family Grows, Infant Jogger Grows With You
10065	309	HRO	Prevents swimmer's ear.	<BYTE value>	Swimmers Can Prevent Ear Infection All Season Long
10066	309	SHIM	Extra-gentle formula. Can be used every day for prevention or treatment of swimmer's ear.	<BYTE value>	Swimmer's Ear Drops Specially Formulated for Children
10067	310	SHIM	Blue heavy-duty foam board with Shimara or team logo.	<BYTE value>	Exceptionally Durable, Compact Kickboard for Team Practice
10068	310	ANZ	White. Standard size.	<BYTE value>	High-Quality Kickboard
10069	311	SHIM	Swim gloves. Webbing between fingers promotes strengthening of arms. Cannot be used in competition.	<BYTE value>	Hot Training Tool - Webbed Swim Gloves Build Arm Strength and Endurance
10070	312	SHIM	Hydrodynamic egg-shaped lens. Ground-in anti-fog elements; Available in blue or smoke.	<BYTE value>	Anti-Fog Swimmer's Goggles: Quantity Discount
10071	312	HRO	Durable competition-style goggles. Available in blue, grey, or white.	<BYTE value>	Swim Goggles: Traditional Rounded Lens For Greater Comfort

catalog Table (7 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10072	313	SHIM	Silicone swim cap. One size. Available in white, silver, or navy. Team Logo Imprinting Available.	<BYTE value>	Team Logo Silicone Swim Cap
10073	313	ANZ	Silicone swim cap. Squared-off top. One size. White.	<BYTE value>	Durable Squared-off Silicone Swim Cap.
10074	302	HRO	Re-usable ice pack. Store in the freezer for instant first-aid. Extra capacity to accommodate water and ice.	<BYTE value>	Water Compartment Combines With Ice to Provide Optimal Orthopedic Treatment

cust_calls Table

customer_num	call_dtime	user_id	call_code	call_descr	res_dtime	res_descr
106	1991-06-12 8:20	maryj	D	Order was received, but two of the cans of ANZ tennis balls within the case were empty.	1991-06-12 8:25	Authorized credit for two cans to customer; issued apology. Called ANZ buyer to report the QA problem.
110	1991-07-07 10:24	richc	L	Order placed one month ago (6/7) not received.	1991-07-07 10:30	Checked with shipping (Ed Smith). Order sent yesterday-we were waiting for goods from ANZ. Next time will call with delay if necessary.
119	1991-07-01 15:00	richc	B	Bill does not reflect credit from previous order.	1991-07-02 8:21	Spoke with Jane Akant in Finance. She found the error and is sending new bill to customer.
121	1991-07-10 14:05	maryj	O	Customer likes our merchandise. Requests that we stock more types of infant joggers. Will call back to place order.	1991-07-10 14:06	Sent note to marketing group of interest in infant joggers.
127	1991-07-31 14:30	maryj	I	Received Hero watches (item # 304) instead of ANZ watches.		Sent memo to shipping to send ANZ item 304 to customer and pickup HRO watches. Should be done tomorrow, 8/1.
116	1990-11-28 13:34	mannyn	I	Received plain white swim caps (313 ANZ) instead of navy with team logo (313 SHM).	1990-11-28 16:47	Shipping found correct case in warehouse and express mailed it in time for swim meet.
116	1990-12-21 11:24	mannyn	I	Second complaint from this customer! Received two cases right-handed outfielder gloves (I HRO) instead of one case lefties.	1990-12-27 08:19	Memo to shipping (Ava Brown) to send case of left-handed gloves; pick up wrong case; memo to billing requesting 5% discount to placate customer due to second offense and lateness of resolution because of holiday.

manufact Table

manu_code	manu_name	lead_time
ANZ	Anza	5
HSK	Husky	5
HRO	Hero	4
NRG	Norge	7
SMT	Smith	3
SHM	Shimara	30
KAR	Karsten	21
NKL	Nikolus	8
PRC	ProCycle	9

state Table

code	sname	code	sname
AK	Alaska	MT	Montana
AL	Alabama	NE	Nebraska
AR	Arkansas	NC	North Carolina
AZ	Arizona	ND	North Dakota
CA	California	NH	New Hampshire
CT	Connecticut	NJ	New Jersey
CO	Colorado	NM	New Mexico
D.C.	DC	NV	Nevada
DE	Delaware	NY	New York
FL	Florida	OH	Ohio
GA	Georgia	OK	Oklahoma
HI	Hawaii	OR	Oregon
IA	Iowa	PA	Pennsylvania
ID	Idaho	PR	Puerto Rico
IL	Illinois	RI	Rhode Island
IN	Indiana	SC	South Carolina
KS	Kansas	SD	South Dakota
KY	Kentucky	TN	Tennessee
LA	Louisiana	TX	Texas
MA	Massachusetts	UT	Utah
MD	Maryland	VA	Virginia
ME	Maine	VT	Vermont
MI	Michigan	WA	Washington
MN	Minnesota	WI	Wisconsin
MO	Missouri	WV	West Virginia
MS	Mississippi	WY	Wyoming

System Catalog

In This Chapter	2-3
Using the System Catalog	2-4
Accessing the System Catalog	2-8
Updating System Catalog Data	2-9
Structure of the System Catalog	2-9
SYSBLOBS	2-10
SYSCHECKS	2-11
SYSCOLAUTH	2-11
SYSCOLDEPEND	2-12
SYSCOLUMNS	2-13
SYSCONSTRAINTS	2-16
SYSDEFAULTS	2-17
SYSDEPEND	2-18
SYSINDEXES	2-18
SYSOPCLSTR	2-21
SYSPROCAUTH	2-23
SYSPROCBODY	2-24
SYSPROCEDURES	2-25
SYSPROCPPLAN	2-26
SYSREFERENCES	2-27
SYSSYNONYMS	2-27
SYSSYNTABLE	2-28
SYSTABAUTH	2-29
SYSTABLES	2-30
SYSUSERS	2-32
SYSVIEWS	2-33

System Catalog Map 2-33

In This Chapter

The system catalog consists of tables that describe the structure of the database. Each system catalog table contains specific information about an element in the database. The system catalog also tracks the views, authorized users, and privileges associated with every table you create.

The system catalog tables are generated automatically when you create a database, and you can query them as you would query any other table in the database. The data for a newly created database and the 21 system catalog tables for that database reside in a common area of the disk called a *dbspace*. If you are using the IBM Informix SE database server, the 19 system catalog tables for a newly created database reside in the *dbname.dbs* directory. All tables that make up the system catalog have the prefix “sys” (for example, the **systables** system catalog table).

This chapter covers the following topics:

- How to access tables in the system catalog
- How to update statistics in the system catalog
- The structure, including the name and data type of each column, of the tables that constitute the system catalog.

Using the System Catalog

The database server accesses the system catalog constantly. Each time an SQL statement is processed, the database server accesses the system catalog to determine system privileges, add or verify table names or column names, and so on. For example, the following CREATE SCHEMA block adds the **customer** system catalog table, with its respective indexes and privileges, to the **stores5** database. This block also adds a view, **california**, that restricts the *view* into the **customer** table to only the first and last name of the customer, the company name, and the phone number of all customers that reside in California.

```
CREATE SCHEMA AUTHORIZATION maryl
CREATE TABLE customer
  (customer_num SERIAL(101), fname CHAR(15), lname CHAR(15), company
  CHAR(20),
   address1 CHAR(20), address2 CHAR(20), city CHAR(15), state CHAR(2),
   zipcode CHAR(5), phone CHAR(18))
GRANT ALTER, ALL ON customer TO cathl WITH GRANT OPTION AS maryl
GRANT SELECT ON CUSTOMER TO public
GRANT UPDATE (fname, lname, phone) ON customer TO nhowe
CREATE VIEW california AS
  SELECT fname, lname, company, phone FROM customer WHERE state = "CA"
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num)
CREATE INDEX state_ix ON customer (state);
```

To process this CREATE SCHEMA block, the database server first accesses the system catalog to verify the following information:

- The new table and view names do not already exist in the database. (If the database is ANSI-compliant, the database server verifies that the table and view names do not already exist for the specified owners.)
- The user has permission to create the tables and grant user privileges.
- The column names in the CREATE VIEW and CREATE INDEX statements exist in the **customer** table.

In addition to verifying this information and creating two *new tables*, the database server adds *new rows* to the following system catalog tables:

- **systables**
- **syscolumns**
- **sysviews**

- **systabauth**
- **syscolauth**
- **sysindexes**

The following two new rows of information are added to the **systables** system catalog table after the preceding CREATE SCHEMA block is run.

```

tabname      customer
owner        maryl
partnum      16778361
tabid        101
rowsize      134
ncols        10
nindexes     2
nrows        0
created      04/26/1990
version      1
tabtype      T
locklevel    P
npused       0
fextsize     16
nextsize     16
flags        0
site
dbname

tabname      california
owner        maryl
partnum      0
tabid        102
rowsize      134
ncols        4
nindexes     0
nrows        0
created      04/26/1990
version      0
tabtype      V
locklevel    B
npused       0
fextsize     0
nextsize     0
flags        0
site
dbname

```

Each table recorded in the **systables** system catalog table is assigned a **tabid**, a system-assigned sequential id number that uniquely identifies each table in the database. The system catalog tables receive **tabid** numbers 1 through 21, while the user-created tables receive **tabid** numbers beginning with 100.

The CREATE SCHEMA block also adds 13 rows to the **syscolumns** system catalog table. These rows correspond to the columns in the table **customer** and the view **california**.

colname	tabid	colno	coltype	collength	colmin	colmax
customer_num	101	1	262	4		
fname	101	2	0	15		
lname	101	3	0	15		
company	101	4	0	20		
address1	101	5	0	20		
address2	101	6	0	20		
city	101	7	0	15		
state	101	8	0	2		
zipcode	101	9	0	5		
phone	101	10	0	18		
fname	102	1	0	15		
lname	102	2	0	15		
company	102	3	0	20		
phone	102	4	0	18		

In the **syscolumns** system catalog table, each column within a table is assigned a sequential column number, **colno**, that uniquely identifies the column within its table. For example, the column **fname** in the **customer** table is assigned the **colno 2**, while the column **fname** in the view **california** is assigned the **colno 1**. Note that the **colmin** and **colmax** columns contain no entries. These two columns contain values only when a column has a composite index, has no null or duplicate values, and the UPDATE STATISTICS statement has been run.

The following rows are added to the **sysviews** system catalog table. These rows correspond to the CREATE VIEW portion of the CREATE SCHEMA block.

tabid	seq	viewtext
102	0	create view "mary1".california (customer_num, fname, lname, company
102	1	,address1, address2, city, state, zipcode, phone) as select x0.custom
102	2	er_num, x0.fname, x0.lname, x0.company, x0.address1, x0.address2
102	3	,x0.city, x0.state, x0.zipcode, x0.phone from "mary1".customer
102	4	x0 where (x0.state = 'CA');

The **sysviews** system catalog table contains the CREATE VIEW statement used to create the view. Each line of the CREATE VIEW statement in the current schema is stored in this table. In the **viewtext** column, the **x0** that precedes the column names in the statement (for example, **x0.fname**) operates as an alias name that distinguishes between the same columns used in a self-join.

The CREATE SCHEMA block also adds the following rows to the **sysstabauth** system catalog table. These rows correspond to the user privileges granted on **customer** and **california**.

grantor	grantee	tabid	tabauth
maryl	public	101	su-idx--
maryl	cathl	101	SU-IDXAR
maryl	nhowe	101	---*-----
	maryl	102	SU-ID---

The **tabauth** column of this table specifies the table-level privileges granted to users on the **customer** and **california** tables. This column uses an 8-byte pattern—**s** (select), **u** (update), ***** (column-level privilege), **i** (insert), **d** (delete), **x** (index), **a** (alter), **r** (references)—to identify the type of privilege granted. In this example, the user **nhowe** has column-level privileges on the **customer** table.

In addition, three rows are added to the **syscolauth** system catalog table. These rows correspond to the user privileges granted on specific columns in the **customer** table.

grantor	grantee	tabid	colno	colauth
maryl	nhowe	101	2	-u-
maryl	nhowe	101	3	-u-
maryl	nhowe	101	10	-u-

The **colauth** column of this table specifies the column-level privileges granted on the **customer** table. This column uses a 3-byte pattern—**s** (select), **u** (update), **r** (references)—to identify the type of privilege granted. For example, the user **nhowe** has update privileges on the second column (**colno2**) of the **customer** table.

The CREATE SCHEMA block adds two rows to the **sysindexes** system catalog table. These rows correspond to the indexes created on the **customer** table.

idxname	c_num_ix	state_ix
owner	maryl	maryl
tabid	101	101
idxtype	U	D
clustered		
part1	1	8
part2	0	0
part3	0	0
part4	0	0
part5	0	0
part6	0	0
part7	0	0
part8	0	0
part9	0	0
part10	0	0
part11	0	0
part12	0	0
part13	0	0
part14	0	0
part15	0	0
part16	0	0
levels		
leaves		
nunique		
clust		

In this table, the **idxtype** column identifies whether the index created is unique or a duplicate. For example, the index **c_num_ix** placed on the **customer_num** column of the **customer** table is unique.

Accessing the System Catalog

Normal user access to the system catalog is read-only. Users with Connect or Resource privileges cannot alter the system catalog. They can, however, access data in the system catalog tables on a read-only basis using standard SELECT statements. For example, the following SELECT statement displays all the table names and corresponding table id numbers of user-created tables in the database:

```
SELECT tabname, tabid FROM systables WHERE tabid > 99
```

Warning: Although the user “informix” can modify the system catalog tables, we strongly recommend that you do not update, delete, or alter any rows in them. Modifying the system catalog tables can destroy the integrity of the database.



Updating System Catalog Data

The optimizer in Informix database servers determines the most efficient strategy for executing SQL queries. This allows you to query the database without having to fully consider which tables to search first in a join or which indexes to use. The optimizer uses information from the system catalog to determine the best possible query strategy.

By using the UPDATE STATISTICS statement to update the system catalog, you can optimize these database-searching strategies. When you delete a number of rows from a table or modify a table, the database server does not automatically update the related statistical data in the system catalog. For example, if you delete a number of rows in a table using the DELETE statement, the **nrows** column in the **systables** system catalog table, which holds the number of rows for that table, is not updated. The UPDATE STATISTICS statement causes the database server to recalculate data in the **systables**, **syscolumns**, and **sysindexes** system catalog tables. After you run UPDATE STATISTICS, the **systables** system catalog table holds the correct value in its **nrows** column.

Whenever you perform extensive modifications to a table, use the UPDATE STATISTICS statement to update data in the system catalog. For more information on the UPDATE STATISTICS statement, see page [7-335](#) in this manual.

Structure of the System Catalog

The following system catalog tables describe the structure of the Informix database:

- **sysblobs**
- **syschecks**
- **syscolauth**
- **syscoldepend**
- **syscolumns**
- **sysconstraints**
- **sysdefaults**
- **sysdepend**
- **sysindexes**

- **sysopclstr**
- **sysprocauth**
- **sysprocbody**
- **sysprocedures**
- **sysprocplan**
- **sysreferences**
- **syssynonyms**
- **syssyntable**
- **systabauth**
- **systables**
- **sysusers**
- **sysviews**

SYSBLOBS

The **sysblobs** system catalog table specifies the storage location of a blob column. It contains one row for each blob column in a table. The **sysblobs** system catalog table has the following columns.

Column Name	Type	Explanation
spacename	CHAR(18)	blob space, db space, or family name
type	CHAR(1)	media type: M = magnetic O = optical
tabid	INTEGER	table identifier
colno	SMALLINT	column number

A composite index for the **tabid** and **colno** columns allows only unique values.

SYSCHECKS

The **syschecks** system catalog table describes each check constraint defined in the database. Since the **syschecks** system catalog table stores both the ASCII text and binary encoded form of the check constraint, it contains multiple rows for each check constraint. The **syschecks** system catalog table has the following columns.

Column Name	Type	Explanation
constrid	INTEGER	constraint identifier
type	CHAR(1)	form in which the check constraint is stored: B = binary encoded T = ASCII text
seqno	SMALLINT	line number of the check constraint
checktext	CHAR(32)	text of the check constraint

A composite index for the **constrid**, **type**, and **seqno** columns allows only unique values.

SYSCOLAUTH

The **syscolauth** system catalog table describes each set of privileges granted on a column. It contains one row for each set of column privileges granted in the database. The **syscolauth** system catalog table has the following columns.

Column Name	Type	Explanation
grantor	CHAR(8)	grantor of privilege
grantee	CHAR(8)	grantee (receiver) of privilege

(1 of 2)

Column Name	Type	Explanation
tabid	INTEGER	table identifier
colno	SMALLINT	column number
colauth	CHAR(3)	3-byte pattern that specifies column privileges: s = select u = update r = references

(2 of 2)

If the **colauth** privilege code is uppercase (for example, S for select), the user who is granted this privilege can also grant it to others. If the **colauth** privilege code is lowercase (for example, s for select), the user who is granted this privilege cannot grant it to others.

A composite index for the **tabid**, **grantor**, **grantee**, and **colno** columns allows only unique values. A composite index for the **tabid** and **grantee** columns allows duplicate values.

SYSCOLDEPEND

The **syscoldepend** system catalog table tracks the table columns specified in each check constraint. Since a check constraint can involve more than one column in a table, it can contain multiple rows for each check constraint. The **syscoldepend** system catalog table has the following columns.

Column Name	Type	Explanation
constrid	INTEGER	constraint identifier
tabid	INTEGER	table identifier
colno	SMALLINT	column number

A composite index for the **constrid**, **tabid**, and **colno** columns allows only unique values. A composite index for the **tabid** and **colno** columns allows duplicate values.

SYSCOLUMNS

The **syscolumns** system catalog table describes each column in the database. There is one row for each column defined in a table or view. If you are using the IBM Informix OnLine database server, the **syscolumns** system catalog table has the following columns.

Column Name	Type	Explanation
colname	CHAR(18)	column name
tabid	INTEGER	table identifier
colno	SMALLINT	column number sequentially assigned by the system (ordinally from left to right within each table)
coltype	SMALLINT	code for column data type: 0 = CHAR 7 = DATE 1 = SMALLINT 8 = MONEY 2 = INTEGER 10 = DATETIME 3 = FLOAT 11 = BYTE 4 = SMALLFLOAT 12 = TEXT 5 = DECIMAL 13 = VARCHAR 6 = SERIAL 14 = INTERVAL
collength	SMALLINT	column length (in bytes)
colmin	INTEGER	second minimum value
colmax	INTEGER	second maximum value

If the **coltype** column contains a value greater than 256, it does not allow null values. To determine the data type for a **coltype** column that contains a value greater than 256, subtract 256 from the value and evaluate the remainder based on the possible **coltype** values. For example, if a column has a **coltype** value of 262, subtracting 256 leaves a remainder of 6, which indicates that this column uses a SERIAL data type.

The value that the **collength** column holds depends on the data type of the column. If the data type of the column is BYTE or TEXT, **collength** holds the length of the descriptor. A **collength** value for a MONEY or DECIMAL column is determined using the following formula:

$$(\textit{precision} * 256) + \textit{scale}$$

For columns of type VARCHAR, the *max-size* and *min-space* values are encoded in the **collength** column using the following formula:

$$(\textit{min-space} * 256) + \textit{max-size}$$

For columns of type DATETIME or INTERVAL, **collength** is determined using the following formula:

$$(\textit{length} * 256) + (\textit{largest_qualifier value} \times 16) + \textit{smallest_qualifier value}$$

The *length* is the physical length of the DATETIME or INTERVAL field, and the *largest_qualifier* and *smallest_qualifier* have the following values.

Field Qualifier	Value
YEAR	0
MONTH	2
DAY	4
HOUR	6
MINUTE	8
SECOND	10
FRACTION(1)	11
FRACTION(2)	12
FRACTION(3)	13
FRACTION(4)	14
FRACTION(5)	15

For example, if a DATETIME YEAR TO MINUTE column has a length of 12 (such as YYYY:DD:MM:HH:MM), a *largest_qualifier value* 0 (for YEAR), and a *smallest_qualifier value* 8 (for MINUTE), the **collength** value is 3080 ((256 * 12) + (0 * 16) + 8).

The **colmin** and **colmax** column values hold the second smallest and second largest data values in the column. For example, if the values in an indexed column are 1, 2, 3, 4, and 5, 2 is the **colmin** value and 4 is the **colmax** value. Storing the second smallest and second largest data values allows the database server to make assumptions about the range of values in a given column and, in turn, further optimize searching strategies. The **colmin** and **colmax** columns contain values only if the column is indexed and you have run UPDATE STATISTICS. If you store BYTE or TEXT data in the **tblspace**, the **colmin** value is -1. The values for all other noninteger column types are the initial four bytes of the maximum or minimum value, treated as an integer.

A composite index for the **tabid** and **colno** columns allows only unique values.

If you are using the IBM Informix SE database server, the **syscolumns** system catalog table has the following columns.

Column Name	Type	Explanation
colname	CHAR(18)	column name
tabid	INTEGER	table identifier
colno	SMALLINT	column number sequentially assigned by the system (ordinally from left to right within each table)
coltype	SMALLINT	code for column data type: 0 = CHAR 5 = DECIMAL 1 = SMALLINT 6 = SERIAL 2 = INTEGER 7 = DATE 3 = FLOAT 8 = MONEY 4 = SMALLFLOAT 10 = DATETIME 14 = INTERVAL
collength	SMALLINT	column length (in bytes)

A composite index for the **tabid** and **colno** columns allows only unique values.

SYSCONSTRAINTS

The **sysconstraints** system catalog table lists the constraints placed on the columns in each database table. An entry also is placed in the **sysindexes** system catalog table for each unique constraint, primary-key constraint, or referential constraint you create, if the constraint does not already have a corresponding entry in the **sysindexes** system catalog table. (Since indexes can be shared, more than one constraint can be associated with an index.) The **sysconstraints** system catalog table has the following columns.

Column Name	Type	Explanation
constrid	SERIAL	system-assigned sequential identifier
constrname	CHAR(18)	constraint name
owner	CHAR(8)	user name of owner
tabid	INTEGER	table identifier
constrtype	CHAR(1)	specifies constraint type C = check constraint P = primary key R = referential U = unique
idxname	CHAR(18)	index name

A composite index for the **constrname** and **owner** columns allows only unique values. The index for the **tabid** column allows duplicate values, while the index for the **constrid** column allows only unique values.

SYSDEFAULTS

The **sysdefaults** system catalog table lists the user-defined defaults placed on each column in the database. There is one row for each user-defined default value. If a default is not explicitly specified, there is no entry in this table. The **sysdefaults** system catalog table has the following columns.

Column Name	Type	Explanation
tabid	INTEGER	table identifier
colno	SMALLINT	column identifier
type	CHAR(1)	default type: L = literal default U = USER C = CURRENT N = NULL T = TODAY S = DBSERVERNAME
default	CHAR(256)	If default type = L, the literal default value

If a literal is specified for the default value, it is stored in the **default** column as ASCII text. If the literal value is not of type CHAR, the **default** column consists of two parts. The first part is the six-bit representation of the binary value of the default value structure. The second part is the default value in English text. The two parts are separated by a space.

If the data type of the column is not CHAR or VARCHAR, a binary representation is encoded in the **default** column.

A composite index for both the **tabid** and **colno** columns allows only unique values.

SYSDEPEND

The **sysdepend** system catalog table describes how each view or table depends on other views or tables. There is one row in this table for each dependency, so a view based on three tables has three rows. The **sysdepend** system catalog table has the following columns.

Column Name	Type	Explanation
btabid	INTEGER	tabid of base table or view
btype	CHAR(1)	base object type T = table V = view
dtabid	INTEGER	tabid of dependent table
dtype	CHAR(1)	dependent object type (V = view); only view is currently implemented

The **btabid** and **dtabid** columns are indexed and allow duplicate values.

SYSINDEXES

The **sysindexes** system catalog table describes the indexes in the database. It contains one row for each index defined in the database. The **sysindexes** system catalog table has the following columns.

Column Name	Type	Explanation
idxname	CHAR(18)	index name
owner	CHAR(8)	owner of index ("informix" for system catalog tables and user name for database tables)
tabid	INTEGER	table identifier
idxtype	CHAR(1)	index type U = unique D = duplicates

(1 of 2)

Column Name	Type	Explanation
clustered	CHAR(1)	clustered or nonclustered index (C = clustered)
part1	SMALLINT	column number of a single index or the 1st component of a composite index
part2	SMALLINT	2nd component of a composite index
part3	SMALLINT	3rd component of a composite index
part4	SMALLINT	4th component of a composite index
part5	SMALLINT	5th component of a composite index
part6	SMALLINT	6th component of a composite index
part7	SMALLINT	7th component of a composite index
part8	SMALLINT	8th component of a composite index
part9	SMALLINT	9th component of a composite index
part10	SMALLINT	10th component of a composite index
part11	SMALLINT	11th component of a composite index
part12	SMALLINT	12th component of a composite index
part13	SMALLINT	13th component of a composite index
part14	SMALLINT	14th component of a composite index
part15	SMALLINT	15th component of a composite index
part16	SMALLINT	16th component of a composite index
levels	SMALLINT	number of B+ tree levels
leaves	INTEGER	number of leaves
nunique	INTEGER	number of unique keys
clust	INTEGER	degree of clustering: smaller numbers correspond to greater clustering

(2 of 2)

Changes that affect existing indexes are reflected in this table only after you run UPDATE STATISTICS.

Each **part n** column component of a composite index (the **part1** through **part16** columns in this table) holds the column number (**colno**) of each part of the 16 possible parts of a composite index.

The **tabid** column is indexed and allows duplicate values. A composite index for the **idxname**, **owner**, and **tabid** columns allows only unique values.

If you are using the IBM Informix SE database server, the **sysindexes** system catalog table has the following columns.

Column Name	Type	Explanation
idxname	CHAR(18)	index name
owner	CHAR(8)	owner of index ("informix" for system tables and user name for database tables)
tabid	INTEGER	table identifier
idxtype	CHAR(1)	index type U = unique D = duplicates
clustered	CHAR(1)	clustered or nonclustered index (C = clustered)
part1	SMALLINT	column number of a single index or the 1st component of a composite index
part2	SMALLINT	2nd component of a composite index
part3	SMALLINT	3rd component of a composite index
part4	SMALLINT	4th component of a composite index
part5	SMALLINT	5th component of a composite index

(1 of 2)

Column Name	Type	Explanation
part6	SMALLINT	6th component of a composite index
part7	SMALLINT	7th component of a composite index
part8	SMALLINT	8th component of a composite index

(2 of 2)

Each **part n th** column component of a composite index (the **part1** through **part8** columns in this table) holds the column number (**colno**) of each part of the 8 possible parts of an index.

The **tabid** column is indexed and allows duplicate values. A composite index for both the **idxname** and **tabid** columns allows only unique values.

SYSOPCLSTR

The **sysopclstr** system catalog table defines each optical cluster in the database. It contains one row for each optical cluster. The **sysopclstr** system catalog table has the following columns.

Column Name	Type	Explanation
owner	CHAR(8)	owner of the cluster
clstrname	CHAR(18)	name of the cluster
clstrsize	INTEGER	size of the cluster
tabid	INTEGER	table identifier
blobcol1	SMALLINT	blob column number 1
blobcol2	SMALLINT	blob column number 2
blobcol3	SMALLINT	blob column number 3
blobcol4	SMALLINT	blob column number 4
blobcol5	SMALLINT	blob column number 5
blobcol6	SMALLINT	blob column number 6

(1 of 3)

Column Name	Type	Explanation
blobcol7	SMALLINT	blob column number 7
blobcol8	SMALLINT	blob column number 8
blobcol9	SMALLINT	blob column number 9
blobcol10	SMALLINT	blob column number 10
blobcol11	SMALLINT	blob column number 11
blobcol12	SMALLINT	blob column number 12
blobcol13	SMALLINT	blob column number 13
blobcol14	SMALLINT	blob column number 14
blobcol15	SMALLINT	blob column number 15
blobcol16	SMALLINT	blob column number 16
clstrkey1	SMALLINT	cluster key number 1
clstrkey2	SMALLINT	cluster key number 2
clstrkey3	SMALLINT	cluster key number 3
clstrkey4	SMALLINT	cluster key number 4
clstrkey5	SMALLINT	cluster key number 5
clstrkey6	SMALLINT	cluster key number 6
clstrkey7	SMALLINT	cluster key number 7
clstrkey8	SMALLINT	cluster key number 8
clstrkey9	SMALLINT	cluster key number 9
clstrkey10	SMALLINT	cluster key number 10
clstrkey11	SMALLINT	cluster key number 11
clstrkey12	SMALLINT	cluster key number 12
clstrkey13	SMALLINT	cluster key number 13

(2 of 3)

Column Name	Type	Explanation
clstrkey14	SMALLINT	cluster key number 14
clstrkey15	SMALLINT	cluster key number 15
clstrkey16	SMALLINT	cluster key number 16

(3 of 3)

A composite index for both the **clstrname** and **owner** columns allows only unique values. The **tabid** column allows duplicate values.

SYSPROCAUTH

The **sysprocauth** table describes the privileges granted on a procedure. It contains one row for each set of privileges granted. The **sysprocauth** system catalog table has the following columns.

Column Name	Type	Explanation
grantor	CHAR(8)	grantor of procedure
grantee	CHAR(8)	grantee (receiver) of procedure
procid	INTEGER	procedure identifier
procauth	CHAR(1)	type of procedure permission granted: e = execute permission on procedure E = execute permission and the ability to grant it to others

A composite index for the **procid**, **grantor**, and **grantee** columns allows only unique values. The composite index for the **procid** and **grantee** columns allows duplicate values.

SYSPROCBODY

The **sysprocbody** system catalog table describes the compiled version of each stored procedure in the database. Since the **sysprocbody** system catalog table stores the text of the procedure, there can be multiple rows for each procedure. The **sysprocbody** system catalog table has the following columns.

Column Name	Type	Explanation
procid	INTEGER	procedure identifier
datakey	CHAR(1)	data descriptor type: D = user document text T = actual procedure source R = return value type list S = procedure symbol table L = constant procedure data string (i.e, literal numbers or quoted strings) P = interpreter instruction code
seqno	INTEGER	line number of the procedure
data	CHAR(256)	actual text of the procedure

While the **datakey** column indicates the type of data stored, the **data** column contains the actual data, which can be one of the following: the encoded return values list, the encoded symbol table, constant data, compiled code for the procedure, or the text of the procedure and its documentation.

A composite index for the **procid**, **datakey**, and **seqno** columns allows only unique values.

SYSPROCEDURES

The **sysprocedures** system catalog table lists the characteristics for each stored procedure in the database. It contains one row for each procedure. The **sysprocedures** system catalog table has the following columns.

Column Name	Type	Explanation
procname	CHAR(18)	procedure name
owner	CHAR(8)	owner name
procid	SERIAL	procedure identifier
mode	CHAR(1)	mode type: D = DBA O = OWNER
retsize	INTEGER	compiled size (in bytes) of values
symsize	INTEGER	compiled size (in bytes) of symbol table
datasize	INTEGER	compiled size (in bytes) constant data
codesize	INTEGER	compiled size (in bytes) of procedure instruction code
numargs	INTEGER	number of procedure arguments

A composite index for the **procname** and **owner** columns allows only unique values.

SYSPROCPLAN

The **sysprocplan** system catalog table describes the query execution plans and dependency lists for DML statements within each stored procedure. Since different parts of a procedure plan can be created on different dates, the table can contain multiple rows for each procedure. The **sysprocplan** system catalog table has the following columns.

Column Name	Type	Description
procid	INTEGER	procedure identifier
planid	INTEGER	plan identifier
datakey	CHAR(1)	identifier procedure plan part: D = dependency list Q = execution plan
seqno	INTEGER	line number of plan
created	DATE	date plan created
datasize	INTEGER	size (in bytes) of the list or plan
data	CHAR(256)	encoded (compiled) list or plan

A composite index for the **procid**, **planid**, **datakey**, and **seqno** columns allows only unique values.

SYSREFERENCES

The **sysreferences** system catalog table lists the referential constraints placed on columns in the database. It contains a row for each referential constraint in the database. The **sysreferences** table has the following columns.

Column Name	Type	Explanation
constrid	INTEGER	constraint identifier
primary	INTEGER	constrid of the corresponding primary key
ptabid	INTEGER	tabid of the primary key
updrule	CHAR(1)	Reserved for future use; displays an R.
delrule	CHAR(1)	Reserved for future use; displays an R.
matchtype	CHAR(1)	Reserved for future use; displays an N.
pendant	CHAR(1)	Reserved for future use; displays an N.

The **constrid** column is indexed and allows only unique values. The **primary** column is indexed and allows duplicate values.

SYSSYNONYMS

The **syssynonyms** system catalog table lists the synonyms for each table or view. It contains a row for every synonym defined in the database. The **syssynonyms** system catalog table has the following columns.

Column Name	Type	Explanation
owner	CHAR(8)	user name of owner
synname	CHAR(18)	synonym identifier
created	DATE	date synonym created
tabid	INTEGER	table identifier



A composite index for the **owner** and **synonym** columns allows only unique values. The **tabid** column is indexed and allows duplicate values.

Tip: *Version 4.0 or later of any IBM Informix product no longer uses this table. However, any **syssynonyms** entries made prior to version 4.0 will remain in this table.*

SYSSYNTABLE

The **syssyntable** system catalog table outlines the mapping between each synonym and the object it represents. It contains one row for each entry in the **syntables** table that has a **tabtype** S. The **syssyntable** system catalog table has the following columns.

Column Name	Type	Explanation
tabid	INTEGER	table identifier
servername	CHAR(18)	server name
dbname	CHAR(18)	database name
owner	CHAR(8)	user name of owner
tablename	CHAR(18)	name of table
btabid	INTEGER	tabid of base table or view

If you define a synonym for a table that is in your current database, only the **tabid** and **btabid** columns are used. If you define a synonym for a table that is external to your current database, the **btabid** column is not used, but the **tabid**, **servername**, **dbname**, **owner**, and **tablename** columns are used.

An index for the **tabid** column allows only unique values. The **btabid** column is indexed to allow duplicate values.

If you are using the IBM Informix SE database server, only the **tabid** and **btabid** columns are used.

SYSTABAUTH

The **systabauth** system catalog table describes each set of privileges granted in a table. It contains one row for each set of table privileges granted in the database. The **systabauth** system catalog table has the following columns.

Column Name	Type	Explanation
grantor	CHAR(8)	grantor of privilege
grantee	CHAR(8)	grantee (receiver) of privilege
tabid	INTEGER	table identifier
tabauth	CHAR(8)	8-byte pattern that specifies table privileges: s = select u = update * = column-level authority i = insert d = delete x = index a = alter r = references

If the **tabauth** privilege code is uppercase (for example, S for select), the user who is granted this privilege also can grant it to others. If the **tabauth** privilege code is lowercase (for example, s for select), the user who is granted this privilege cannot grant it to others.

A composite index for the **tabid**, **grantor**, and **grantee** columns allows only unique values. The composite index for the **tabid** and **grantee** columns allows duplicate values.

SYSTABLES

The **sysables** system catalog table describes each table in the database. It contains one row for each table, view, or synonym defined in the database. This includes all database tables and the system catalog tables themselves. If you are using the IBM Informix OnLine database server, the **sysables** system catalog table has the following columns.

Column Name	Type	Explanation
tablename	CHAR(18)	name of table
owner	CHAR(8)	owner of table ("informix" for system catalog tables and user name for database tables)
partnum	INTEGER	tblspace identifier (similar to tabid)
tabid	SERIAL	system-assigned sequential ID number (system tables: 1-21, user tables: 100-nnn)
rowsize	SMALLINT	row size
ncols	SMALLINT	number of columns
nindexes	SMALLINT	number of indexes
nrows	INTEGER	number of rows
created	DATE	date created
version	INTEGER	number of times table was altered
tabtype	CHAR(1)	table type T = table V = view P = private synonym P = synonym (in an ANSI-compliant database) S = synonym

(1 of 2)

Column Name	Type	Explanation
locklevel	CHAR(1)	lock mode for a table B = page P = page R = row
npused	INTEGER	number of data pages in use
fextsize	INTEGER	size of initial extent (kilobytes)
nextsize	INTEGER	size of all subsequent extents (kilobytes)
flags	SMALLINT	reserved for future use
site	CHAR(18)	reserved for future use
dbname	CHAR(18)	reserved for future use

(2 of 2)

The **tabid** column is indexed and must contain unique values. A composite index for both the **tablename** and **owner** columns allows only unique values.

If you are using the IBM Informix SE database server, the **systables** system catalog table has the following columns.

Column Name	Type	Explanation
tablename	CHAR(18)	name of table
owner	CHAR(8)	owner of table ("informix" for system tables and user name for database tables)
dirpath	CHAR(64)	directory path for the table file
tabid	SERIAL	system assigned sequential ID number (system tables: 1-19, user tables: 100-nnn)
rowsize	SMALLINT	row size
ncols	SMALLINT	number of columns
nindexes	SMALLINT	number of indexes

(1 of 2)

Column Name	Type	Explanation
nrows	INTEGER	number of rows
created	DATE	date created
version	INTEGER	number of times table was altered
tabtype	CHAR(1)	table type T = table V = view S = synonym L = log P = private synonym
audpath	CHAR(64)	audit filename (full pathname)

(2 of 2)

The **tabid** column is indexed and must contain unique values. A composite index for the **tablename** and **owner** columns allows only unique values.

SYSUSERS

The **sysusers** system catalog table describes each set of privileges granted in the database. It contains one row for each user who is granted privileges in the database. The **sysusers** system catalog table has the following columns.

Column Name	Type	Explanation
username	CHAR(8)	name of the database user
usertype	CHAR(1)	specifies database-level privileges: D = DBA (all privileges) R = Resource (create permanent tables and indexes) C = Connect (work within existing tables)
priority	SMALLINT	reserved for future use
password	CHAR(8)	reserved for future use

The **username** column is indexed and allows only unique values.

SYSVIEWS

The **sysviews** system catalog table describes each view defined in the database. Since the **sysviews** system catalog table stores the actual SELECT statement used to create the view, it can contain multiple rows for each view into the database. The **sysviews** system catalog table has the following columns.

Column Name	Type	Explanation
tabid	INTEGER	table identifier
seqno	SMALLINT	line number of the SELECT statement
viewtext	CHAR(64)	actual SELECT statement used to create the view

A composite index for the **tabid** and **seqno** columns allows only unique values.

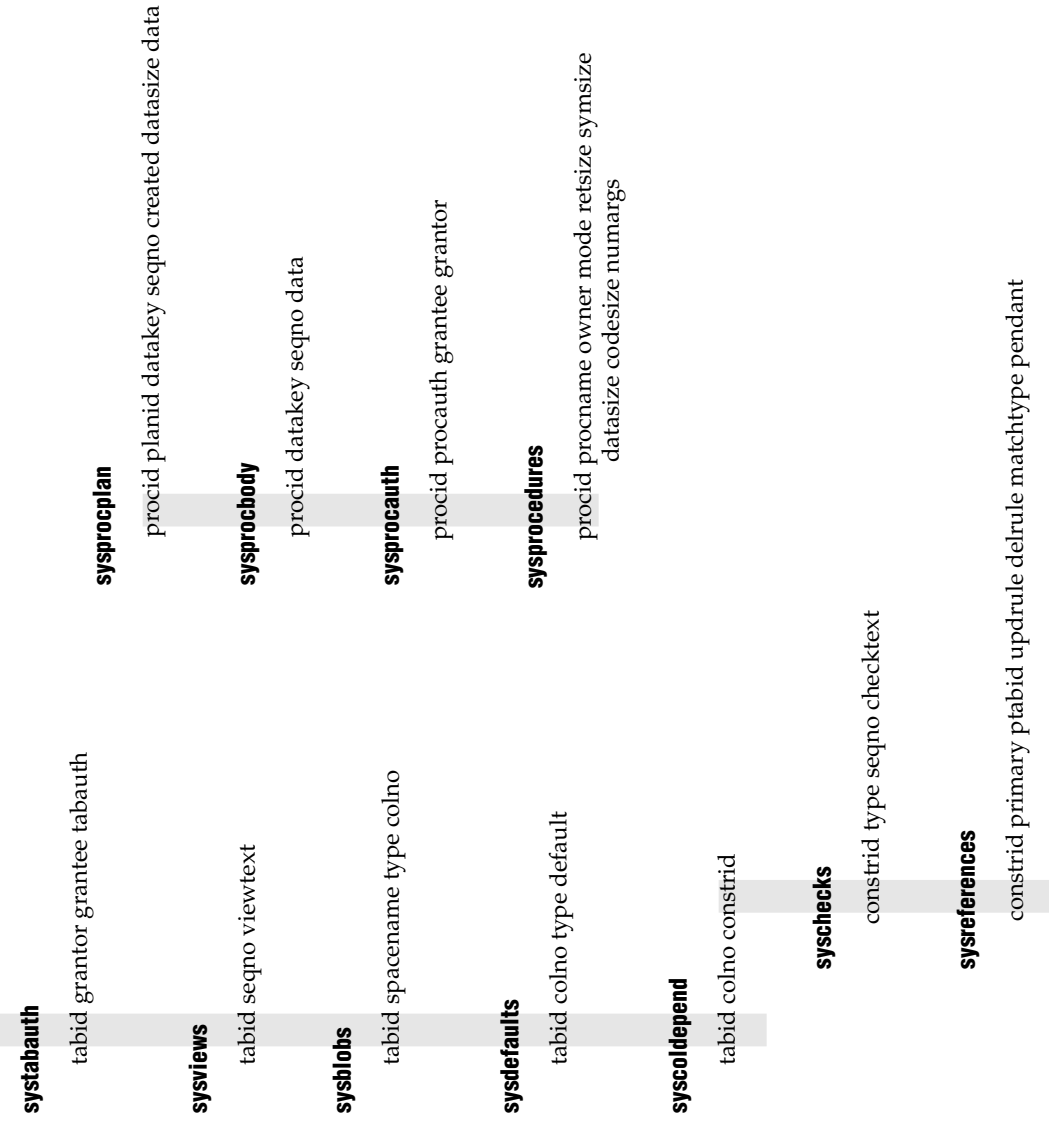
System Catalog Map

[Figure 2-1](#) displays the column names of the tables in an IBM Informix OnLine system catalog. The lines connecting a column in one table to a column in another table indicate columns that contain the same information.

sysables	tabid tabname owner dirpath rowsize ncols nindexes nrows created version tabtype audpath partnum locklevel npused fextsize nextsize flags site dbname
sysstable	tabid tabname servername dbname owne btabid
sysstynoims	tabid owner synname created
syscolumns	tabid colno colname coltype collength colmin colmax
syscolauth	tabid colno grantor grantee colauth
sysindexes	tabid idxname owner idxtype clustered part1—part16 leaves leaves munique clust
sysconstraints	tabid idxname constrid constrname owner constrtype
sysopclstr	tabid owner clstrname clstrsize blobcol1—blobcol16 clstrike1—clstrike16
sysusers	username usertype priority password

Figure 2-1
System catalog map (1 of 2)

Figure 2-1 (continued)
System catalog map (2 of 2)



Data Types

In This Chapter	3-3
Database Data Types	3-4
BYTE	3-5
CHAR(n)	3-6
CHARACTER(n)	3-7
DATE	3-7
DATETIME	3-8
DEC	3-11
DECIMAL[(p,s)]	3-11
DOUBLE PRECISION(n)	3-12
FLOAT(n).	3-12
INT	3-13
INTEGER.	3-13
INTERVAL	3-13
MONEY(p,s).	3-17
NUMERIC(p,s)	3-17
REAL	3-17
SERIAL(n)	3-18
SMALLFLOAT	3-19
SMALLINT	3-19
TEXT	3-19
VARCHAR(m,r)	3-21
Data Type Conversions	3-22
Converting from Number to Number	3-23
Converting Between Number and CHAR.	3-24
Converting Between DATE and DATETIME.	3-24

Range of Operations Using DATE, DATETIME, and INTERVAL . . .	3-25
Manipulating DATETIME Values	3-26
Manipulating DATETIME with INTERVAL Values	3-27
Manipulating DATE with DATETIME and INTERVAL Values. . .	3-28
Manipulating INTERVAL Values	3-30
Multiplying or Dividing INTERVAL Values	3-30

In This Chapter

Every column in a table is assigned a *data type*. The data type precisely defines the type of values that you can store in that column.

You assign data types with the CREATE TABLE statement and change them with the ALTER TABLE statement. When you change an existing data type, all data is converted to the new data type, if possible. For more information on the ALTER TABLE and CREATE TABLE statements and data type syntax conventions, refer to [Chapter 7, “Syntax.”](#) Refer to the *IBM Informix Guide to SQL: Tutorial* for a detailed discussion of data types.

This chapter covers the following topics:

- Data types supported by IBM Informix products
- Data type conversions
- DATE, DATETIME, and INTERVAL values in arithmetic and relational expressions

Database Data Types

IBM Informix products recognize the data types listed in [Figure 3-1](#).

Figure 3-1
Data types recognized by IBM Informix products

Data Type	Explanation
BYTE	stores any kind of binary data
CHAR	stores any string of letters, numbers, and symbols
CHARACTER	is a synonym for CHAR
DATE	stores calendar date
DATETIME	stores calendar date combined with time of day
DEC	is a synonym for DECIMAL
DECIMAL	stores numbers with definable scale and precision
DOUBLE PRECISION	is a synonym for FLOAT
FLOAT	stores double-precision floating numbers corresponding to the double data type in C
INT	is a synonym for INTEGER
INTEGER	stores whole numbers from -2,147,483,647 to +2,147,483,647
INTERVAL	stores span of time
MONEY	stores currency amount
NUMERIC	is a synonym for DECIMAL
REAL	is a synonym for SMALLFLOAT
SERIAL	stores sequential integers
SMALLFLOAT	stores single-precision floating numbers corresponding to the float data type in C

(1 of 2)

Data Type	Explanation
SMALLINT	stores whole numbers from -32,767 to +32,767
TEXT	stores any kind of text data
VARCHAR	stores character strings of varying length

(2 of 2)

The following sections describe each of these data types.

BYTE

The BYTE data type stores any kind of binary data in an undifferentiated byte stream. Binary data typically consists of saved spreadsheets, program load modules, digitized voice patterns, and so on. The IBM Informix SE database server does not support this data type.

The data type BYTE has no maximum size. A BYTE column has a theoretical limit of 2^{31} bytes and a practical limit determined by your disk capacity.

You can store, retrieve, update, or delete the contents of a BYTE column. However, you cannot use BYTE data items in arithmetic or string operations or assign literals to BYTE items with the SET clause of the UPDATE statement. Nor can you use BYTE items in any of the following ways:

- With aggregate functions
- With the IN clause
- With the MATCHES or LIKE clauses
- With the GROUP BY clause
- With the ORDER BY clause

You only can use BYTE objects in a Boolean expression if you are testing for null values.

You can insert data into BYTE columns in the following ways:

- With the **dbload** or **tbload** utilities
- With the LOAD statement (IBM Informix SQL, IBM Informix 4GL, and DB-Access)

- Through a screen form with the PROGRAM attribute (IBM Informix SQL)
- From BYTE host variables (IBM Informix 4GL and ESQL/C)
- From an embedded SQL program (ESQL/COBOL)

You cannot use a quoted text string, number, or any other actual value to insert or update BYTE columns.

When you select a BYTE column, you can choose to receive all or part of it. To see all of it, use the regular syntax for selecting a column. You also can select any part of a BYTE column by using subscripts as shown in the following example:

```
SELECT cat_picture [1,75] FROM catalog WHERE catalog_num = 10001
```

This statement reads the first 75 bytes of the **cat_picture** column associated with the catalog number 10001.



Tip: If you select a BYTE column using the IBM Informix SQL or DB-Access Interactive Schema Editor, only the phrase <BYTE value> is returned; no actual value is displayed.

CHAR(n)

The CHAR data type stores any string of letters, numbers, and symbols. A character column has a maximum length n , where $1 \leq n \leq 32,767$. (If you are using the IBM Informix SE database server, the maximum length is 32,511.) If you do not specify n , CHAR(1) is assumed.

Character columns typically store names, addresses, phone numbers, and so on. Since the length of this column is fixed, when a character value is retrieved or stored, exactly n bytes of data are transferred. If the value is shorter than n , the string is extended with spaces to make up the n bytes. If the value is longer than n , the string is truncated.

A CHAR value can include tabs and spaces, but no other nonprinting characters.

If you plan to perform calculations on numbers stored in a column, you should assign a number data type to that column. Although you can store numbers in CHAR columns, you might not be able to use them in some arithmetic operations. For example, if you are inserting the sum of values into a character column, you might experience overflow problems if the character column is too small to hold the value. In this case, the insert would fail. However, numbers that have leading zeros (such as some zip codes) will have the zeros stripped if they are stored as number types INTEGER or SMALLINT. You should store these numbers in CHAR columns.

CHAR values are compared to other CHAR values by taking the shorter value and padding it on the right with spaces until the values have equal length. Then, the two values are compared for the full length.

CHAR data types require 1 byte per character, or *n* bytes.

CHARACTER(*n*)

The CHARACTER data type is a synonym for CHAR.

DATE

The DATE data type stores the calendar date. A calendar date is stored internally as an integer value equal to the number of days since December 31, 1899.

The default display format of a DATE column is

mm/dd/yyyy

where *mm* is the month (1-12), *dd* is the day of the month (1-31), and *yyyy* is the year (0001-9999). For month, IBM Informix products accept a number value 1 or 01 for January, and so on. For days, IBM Informix products accept a value 1 or 01 that corresponds to the first day of the month, and so on. If you enter only a two-digit value for year, as in 89 or 90, IBM Informix products assume that the year is in the twentieth century and assign the numbers 1 and 9 (19) as the first two digits of the year.



Since date values are stored as integers, you can use them in arithmetic expressions. For example, you can subtract a DATE value from another DATE value. The result, a positive or negative INTEGER value, indicates the number of days that elapsed between the two dates.

DATE data types require 4 bytes per item.

Tip: You can change the default date format by changing the *DBDATE* environment variable. See [Chapter 4, “Environment Variables,”](#) for more information.

DATETIME

The DATETIME data type stores an instant in time expressed as a calendar date and time of day. You choose how precisely a DATETIME value is stored; its precision can range from a year to a fraction of a second.

The DATETIME data type is composed of a contiguous sequence of fields that represents each component of time you want to record and uses the following syntax:

DATETIME *largest_qualifier* TO *smallest_qualifier*

The *largest_qualifier* and *smallest_qualifier* can be any one of the fields listed in [Figure 3-2](#).

Figure 3-2
DATETIME field qualifiers

Qualifier Field	Valid Entries
YEAR	a year numbered from 1 to 9999 (A.D.)
MONTH	a month numbered from 1 to 12
DAY	a day numbered from 1 to 31, as appropriate to the month in question
HOUR	an hour numbered from 0 (midnight) to 23
MINUTE	a minute numbered from 0 to 59

(1 of 2)

Qualifier Field	Valid Entries
SECOND	a second numbered from 0 to 59
FRACTION	a decimal fraction of a second with up to 5 digits of precision. The default precision is 3 digits (thousandth of a second). Other precisions are indicated explicitly by writing FRACTION(<i>n</i>), where <i>n</i> is the desired number of digits from 1 to 5.

(2 of 2)

A DATETIME column need not include all fields from YEAR to FRACTION; it can include a subset of fields or even a single field. For example, you can enter a value of MONTH TO HOUR into a column that is defined as YEAR TO MINUTE, as long as each entered value contains information for a contiguous sequence of fields. You cannot, however, define a column for just MONTH and HOUR; this entry must include a value for DAY as well.



Tip: *If you are using the IBM Informix SQL or DB-Access TABLE Menu and you do not specify the DATETIME qualifiers, the default DATETIME qualifier, YEAR TO YEAR, is assigned.*

A valid DATETIME literal must include the DATETIME keyword, the values to be entered, and the field qualifiers (see the discussion of literal DATETIME on [page 7-416](#)). You must include these qualifiers because, as noted earlier, the value you are entering can contain fewer fields than defined for that column. Acceptable qualifiers for the first and last fields are identical to the list of valid DATETIME fields displayed in [Figure 3-2](#).

Values for the field qualifiers are written as integers and separated by delimiters. [Figure 3-3](#) lists the delimiters that are used with DATETIME values.

Figure 3-3
DATETIME field delimiters

Delimiter	Placement in DATETIME Expression
hyphen	between the YEAR, MONTH, and DAY portions of the value
space	between the DAY and HOUR portions of the value
colon	between the HOUR and MINUTE and the MINUTE and SECOND portions of the value
decimal point	between the SECOND and FRACTION portions of the value

Figure 3-4 shows a DATETIME YEAR TO FRACTION(3) value with delimiters.

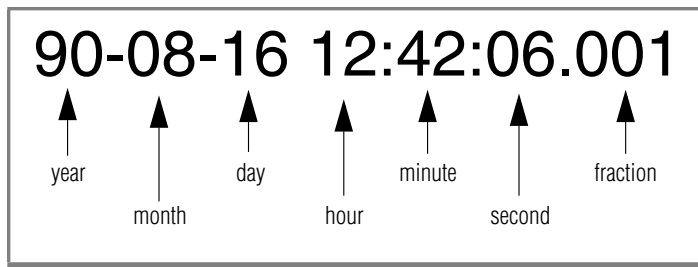


Figure 3-4
Example
DATETIME value
with delimiters

When you enter a value with fewer fields than the defined column, the value you enter is expanded automatically to fill all the defined fields. If you leave out any more-significant fields, that is, fields of larger magnitude than any value you supply, those fields are filled automatically with the current date. If you leave out any less-significant fields, those fields are filled with zeros (or a one for MONTH and DAY) in your entry.

You also can enter DATETIME values as character strings. However, the character string must include information for each field defined in the DATETIME column. For example, the following INSERT statement shows a DATETIME value entered as a character string:

```
INSERT into cust_calls (customer_num, call_dtime, user_id,
    call_code, call_descript)
VALUES (100, "1990-08-14 08:45", "maryj", "D",
    "Order late - placed 6/1/90")
```

In this case, the **call_dtime** column is defined as DATETIME YEAR TO MINUTE. This character string must include values for the year, month, day, hour, and minute fields. If the character string does not contain information for all defined fields (or adds additional fields), the database server returns an error. For more information on entering DATETIME values as character strings, see [“Literal DATETIME” on page 7-416](#).

All fields of a DATETIME column are two-digit numbers except for the year and fraction fields. The year field is stored as four digits. The fraction field requires n digits where $1 \leq n \leq 5$, rounded up to an even number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes required for a DATETIME value:

$$\text{total number of digits for all fields} / 2 + 1$$

For example, a YEAR TO DAY qualifier requires a total of eight digits (four for year, two for month, and two for day). This data value requires five bytes ($8/2 + 1$) of storage.

For information on using DATETIME data in arithmetic and relational expressions, see [“Range of Operations Using DATE, DATETIME, and INTERVAL” on page 3-25](#). For information on using DATETIME as a constant expression, see [“Constant Expressions” on page 7-376](#).

DEC

The DEC data type is a synonym for DECIMAL.

DECIMAL(p,s)

The DECIMAL data type stores decimal floating-point numbers up to a maximum of 32 significant digits, where p is the total number of significant digits (the precision) and s is the number of digits to the right of the decimal point (the scale). When you assign values to both p and s , the decimal variable has fixed-point arithmetic. All numbers with an absolute value less than $0.5 * 10^{-s}$ have the value zero. The largest absolute value of a variable of this type that you can store without an error is $10^{p-s} - 10^{-s}$.

DOUBLE PRECISION(*n*)

Specifying precision and scale parameters is optional. If you do not specify precision (*p*), DECIMAL is treated as DECIMAL(16), a floating decimal with a precision of 16 places. If you do not specify scale (*s*), DECIMAL(*p*) has a precision of *p* and an absolute value range between 10^{-128} and 10^{126} .

A DECIMAL data type column typically stores numbers with fractional parts that must be calculated exactly (for example, rates or percentages). You can use the following formula (rounded up to a whole number of bytes) to calculate the byte storage for a decimal data type:

$$\text{precision} / 2 + 1$$

For example, a DECIMAL data type with a precision of 16 and a scale of 2 (DECIMAL(16,2)) requires 9 bytes ($16/2 + 1$) of storage.

DOUBLE PRECISION(*n*)

The DOUBLE PRECISION data type is a synonym for FLOAT.

FLOAT(*n*)

The FLOAT data type stores double-precision floating-point numbers with up to 16 significant digits. FLOAT corresponds to the double data type in C. The range of values for the FLOAT data type is the same as the range of values for the C double data type on your machine.

You can use *n* to specify the precision of a FLOAT data type, even though SQL ignores the precision. The value *n* must be a whole number between 1 and 14.

A column with the FLOAT data type typically stores scientific numbers that can only be calculated approximately. Since floating-point numbers retain only their most significant digits, the number you enter in this type of column and the number the database server displays can differ slightly. This depends on how your computer stores floating-point numbers internally. For example, you might enter a value of 1.1 into a FLOAT field and, after processing the SQL statement, the database server might display this value as 1.1000001. This occurs when a value has more digits than the floating point number can store. In this case, the value is stored in its approximate form with the least significant digits treated as zeros.

FLOAT data types usually require eight bytes per value.

INT

The INT data type is a synonym for INTEGER.

INTEGER

The INTEGER data type stores whole numbers that range from -2,147,483,647 to 2,147,483,647. The maximum negative number, -2,147,483,648, is a reserved value and cannot be used. The INTEGER data type is stored as a signed binary integer and is typically used to store counts, quantities, and so on.

Arithmetic operations and sort comparisons are performed more efficiently on binary data than on float or decimal data. However, INTEGER columns only can store a limited range of values. If the data value exceeds the numeric range, the database server does not store the value.

INTEGER data types require four bytes per value.

INTERVAL

The INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: *year-month intervals* and *day-time intervals*. A year-month interval can represent a span of years and months, while a day-time interval can represent a span of days, hours, minutes, seconds, and fractions of a second.

An INTERVAL value always is composed of one value, or a contiguous sequence of values, that represent a component of time. An INTERVAL data type is defined using the following syntax:

```
INTERVAL largest_qualifier(n) TO smallest_qualifier(n)
```

where the *largest_qualifier* and *smallest_qualifier* fields both are taken from one of the two INTERVAL classes shown in [Figure 3-5](#), and *n* optionally specifies the precision of the largest field (and smallest field if it is a FRACTION).

Figure 3-5
INTERVAL classes and field qualifiers

YEAR-MONTH INTERVAL Class		DAY-TIME INTERVAL Class	
YEAR	a number of years	DAY	a number of days
MONTH	a number of months	HOUR	a number of hours
		MINUTE	a number of minutes
		SECOND	a number of seconds
		FRACTION	a decimal fraction of a second, with up to 5 digits of precision. The default precision is 3 digits (thousandth of a second). Other precisions are indicated explicitly by writing FRACTION(<i>n</i>), where <i>n</i> is the desired number of digits from 1 to 5.

As with a DATETIME column, you can define an INTERVAL column to include a subset of the fields you need. However, since the INTERVAL data type represents a span of time independent of an actual date, you cannot combine the two INTERVAL classes. For example, since the number of days in a month depends on which month it is, a single INTERVAL data value cannot combine both months and days.

A value entered into an INTERVAL column need not include all fields contained in the column. For example, you can enter a value of HOUR TO SECOND into a column defined as DAY TO SECOND. However, a value must always consist of a contiguous sequence of fields. In the previous example, you cannot enter just HOUR and SECOND values; you must include MINUTE values as well.

A valid INTERVAL literal contains the INTERVAL keyword, the values to be entered, and the field qualifiers (see the discussion of literal INTERVAL on page [page 7-419](#)). When a value contains just one field, the largest and smallest fields are the same.



When you enter a value in an INTERVAL column, you must specify both the largest and smallest fields in the value, just as you do for DATETIME values. In addition, you can use *n* optionally to specify the precision of the first field (and the last field if it is a FRACTION). If the largest and smallest field qualifiers are both FRACTIONS, you can only specify the precision in the last field. Acceptable qualifiers for the largest and smallest fields are identical to the list of INTERVAL fields displayed in [Figure 3-5](#).

Tip: If you are using the IBM Informix SQL or DB-Access TABLE Menu and you do not specify the INTERVAL field qualifiers, the default INTERVAL qualifier, YEAR TO YEAR, is assigned.

The *largest_qualifier* in an INTERVAL value can be up to nine digits long (except for FRACTION, which cannot be more than five digits long), but if the value you wish to enter is greater than the default number of digits allowed for that field, you must explicitly identify the number of significant digits in the value you are entering. For example, to define an INTERVAL of DAY TO HOUR that can store more than 99 days, enter

```
INTERVAL DAY(3) TO HOUR
```

INTERVAL values use the same delimiters as DATETIME values. The delimiters are shown in [Figure 3-6](#).

Figure 3-6
INTERVAL delimiters

Delimiter	Placement in DATETIME Expression
hyphen	between the YEAR and MONTH
space	between the DAY and HOUR portions of the value
colon	between the HOUR and MINUTE and the MINUTE and SECOND portions of the value
decimal point	between the SECOND and FRACTION portions of the value

You also can enter INTERVAL values as character strings. However, the character string must include information for the identical sequence of fields defined for that column. For example, the following INSERT statement shows an INTERVAL value entered as a character string:

```
INSERT INTO manufact (manu_code, manu_name, lead_time)
VALUES ("BRO", "Ball-Racquet Originals", "160")
```

Since the **lead_time** column is defined as INTERVAL DAY(3) TO DAY, this INTERVAL value requires only one field, the span of days required for lead time. Note that if the character string does not contain information for all fields (or adds additional fields), the database server returns an error. For more information on entering INTERVAL values as character strings, see [“Literal INTERVAL” on page 7-419](#).

By default, all fields of an INTERVAL column are two-digit numbers except for the year and fraction fields. The year field is stored as four digits. The fraction field requires n digits where $1 \leq n \leq 5$, rounded up to an even number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes required for an INTERVAL value:

$$\text{total number of digits for all fields} / 2 + 1$$

For example, a YEAR TO MONTH qualifier requires a total of six digits (four for year and two for month). This data value requires four bytes ($6/2 + 1$) of storage.

For information on using INTERVAL data in arithmetic and relational operations, see [“Range of Operations Using DATE, DATETIME, and INTERVAL” on page 3-25](#). For information on using INTERVAL as a constant expression, see [“Literal INTERVAL as an Expression” on page 7-381](#).

MONEY(*p,s*)

The MONEY data type stores currency amounts. Like the DECIMAL data type, the MONEY data type stores fixed-point numbers up to a maximum of 32 significant digits, where *p* is the total number of significant digits (the precision) and *s* is the number of digits to the right of the decimal point (the scale).

Unlike the DECIMAL data type, the MONEY data type always is treated as a fixed-point decimal number. The data type MONEY(*p*) is defined as DECIMAL(*p*,2). If the precision and scale parameters are not specified, MONEY is interpreted as DECIMAL(16,2).

Values in MONEY columns are displayed with a currency symbol (by default, a dollar sign) and a decimal point. You can use the following formula (rounded up to a whole number of bytes) to calculate the byte storage for a MONEY data type:

$$\textit{precision} / 2 + 1$$

For example, a MONEY data type with a precision of 16 and a scale of 2 (MONEY(16,2)) requires 9 bytes ($16/2 + 1$) of storage.



Tip: You can change the display format for money values by changing the *DBMONEY* environment variable. See [Chapter 4, “Environment Variables,”](#) for more information.

NUMERIC(*p,s*)

The NUMERIC data type is a synonym for DECIMAL.

REAL

The REAL data type is a synonym for SMALLFLOAT.

SERIAL(*n*)

The SERIAL data type stores a sequential integer assigned automatically by the database server when a row is inserted. (For more information on inserting values into SERIAL columns, see [“Inserting Values into SERIAL Columns” on page 7-196.](#)) You can define only one SERIAL column in a table.

The default serial starting number is 1, but you can assign an initial value, *n*, when you create or alter the table. You can assign any number greater than 0 as your starting number. The highest serial number you can assign is 2,147,483,647.

Once a serial number is assigned, it cannot be changed. You can, however, insert a value into a serial column (using the INSERT statement) or reset the serial value *n* (using the ALTER TABLE statement), as long as that value does not duplicate any existing values in the table. When you insert a number into a SERIAL column or reset the next value of a SERIAL column, your database server assigns the next number in sequence to the number entered. However, if you reset the next value of a SERIAL column to a value that is less than the values already in that column, the next value is computed using the following formula:

$$\text{maximum existing value in SERIAL column} + 1$$

For example, if you reset the serial value of the **customer_num** column in the **customer** table to 50 and the highest-assigned customer number is 128, the next customer number assigned is 129.

The SERIAL data type is not automatically a unique column. You must apply a unique index to this column to prevent duplicate serial numbers.



Tip: *If you are using the interactive schema editor in IBM Informix SQL or DB-Access to define the table, a unique index is applied automatically to the SERIAL column.*

A SERIAL data column is commonly used to store unique numeric codes (for example, order, invoice, or customer numbers). SERIAL data values require four bytes of storage.

SMALLFLOAT

The SMALLFLOAT data type stores single-precision floating-point numbers with approximately eight significant digits. SMALLFLOAT corresponds to the float data type in C. The range of values for a SMALLFLOAT data type is the same as the range of values for the C float data type on your machine.

A SMALLFLOAT data type column typically stores scientific numbers that can only be calculated approximately. Since floating-point numbers retain only their most significant digits, the number you enter in this type of column and the number the database displays may differ slightly. This depends on how your computer stores floating-point numbers internally. For example, you might enter a value of 1.1 into a SMALLFLOAT field and, after processing the SQL statement, the application tool might display this value as 1.1000001. This occurs when a value has more digits than the floating-point number can store. In this case, the value is stored in its approximate form with the least significant digits treated as zeros.

SMALLFLOAT data types usually require four bytes per value.

SMALLINT

The SMALLINT data type stores small whole numbers that range from -32,767 to 32,767. The maximum negative number, -32,768, is a reserved value and cannot be used. The SMALLINT value is stored as a signed binary integer.

Integer columns typically store counts, quantities, and so on. Since the SMALLINT data type takes up only two bytes per value, arithmetic operations are performed very efficiently. However, this data type stores a limited range of values. If the values exceed the range between the minimum and maximum numbers, the database server does not store the value.

TEXT

The TEXT data type stores any kind of text data. The IBM Informix SE database server does not support this data type.

The data type TEXT has no maximum size. A TEXT column has a theoretical limit of 2^{31} bytes and a practical limit determined by your available disk storage.

TEXT columns typically store memos, manual chapters, business documents, program source files, and so on. A data object of type TEXT can contain a combination of printable ASCII characters and the following control characters:

- Tabs (CTRL-I)
- New lines (CTRL-J)
- New pages (CTRL-L)

You can store, retrieve, update, or delete the contents of a TEXT column. However, you cannot use TEXT data items in arithmetic or string operations, and you cannot assign literals to TEXT items with the SET clause of the UPDATE statement. Nor can you use TEXT items in any of the following ways:

- With aggregate functions
- With the IN clause
- With the MATCHES or LIKE clauses
- With the GROUP BY clause
- With the ORDER BY clause

You only can use TEXT objects in Boolean expressions if you are testing for null values.

You can insert data into TEXT columns in the following ways:

- With the **dbload** or **tbload** utilities
- With the LOAD statement (IBM Informix SQL and IBM Informix 4GL)
- Through a screen form with the PROGRAM attribute (IBM Informix SQL)
- From TEXT host variables (4GL, ESQL/C)
- From an embedded SQL program (ESQL/COBOL)

You cannot use a quoted text string, number, or any other actual value to insert or update TEXT columns.

When you select a TEXT column, you can choose to receive all or part of it. To see all of it, use the regular syntax for selecting a column into a variable. You also can select any part of a TEXT column by using subscripts, as shown in the following example:

```
SELECT cat_descr [1,75] FROM catalog WHERE catalog_num = 10001
```

This statement reads the first 75 bytes of the `cat_descr` column associated with catalog number 10001.

VARCHAR(*m,r*)

The VARCHAR data type stores a character string of varying length, where *m* is the maximum size of the column and *r* is the minimum amount of space reserved for that column. The IBM Informix SE database server does not support this data type.

You must specify the maximum size (*m*) of the VARCHAR column. The size of this parameter can range from 1 to 255 bytes. If you are placing an index on a VARCHAR column, the maximum size is 254 bytes. You can store shorter character strings than the value you specify, but not longer.

Specifying the minimum reserved space (*r*) parameter is optional. This value can range from 0 to 255 bytes but must be less than the maximum size (*m*) of the VARCHAR column. If you do not specify a minimum space value, it defaults to 0. You should specify this parameter when you intend to insert rows with short or null data in this column initially, but expect the data to be updated with longer values later.

While the use of VARCHAR economizes on space used in a table, it has no effect on the size of an index. In an index based on a VARCHAR column, each index key has length *m*, the maximum size of the column.

When you store a VARCHAR value in the database, only its defined characters are stored. The database server does not strip a VARCHAR object of any user-entered trailing blanks, nor does the database server pad the VARCHAR to the full length of the column. However, if you specify a minimum reserved space (*r*) and some of the data values are shorter than that amount, some of the space reserved for rows goes unused.

VARCHAR values are compared to other VARCHAR values and to character values in the same way that character values are compared: the shorter value is padded on the right with spaces until the values have equal lengths. Then they are compared for the full length.

Data Type Conversions

You might want to change the data type of a column when you need to store larger values than the current data type can accommodate. For example, if you create a `SMALLINT` column and find later that you need to store integers larger than 32,768, you must change the data type of that column to store the larger value. You can use the `ALTER TABLE` statement to change the data type of that column.

If you change data types, the new data type must be able to store all the old values. For example, if you convert a column from the `INTEGER` data type to the `SMALLINT` data type and the following values exist in the `INTEGER` column, the database server does not change the data type because `SMALLINT` columns cannot accommodate numbers greater than 32,768.

```
100  400  700  50000  700
```

The same situation can occur if you attempt to transfer data from `FLOAT` or `SMALLFLOAT` columns to `INTEGER`, `SMALLINT`, or `DECIMAL` columns.

Converting from Number to Number

When you convert columns from one number data type to another, you occasionally can find rounding errors. [Figure 3-7](#) indicates which numeric data type conversions are acceptable and what kinds of errors you can encounter when you convert between certain numeric data types.

Figure 3-7
Numeric data type conversion chart

From:	To:				
	SMALLINT	INTEGER	SMALLFLOAT	FLOAT	DECIMAL
SMALLINT	ok	ok	ok	ok	O
INTEGER	X	ok	X	ok	O
SMALLFLOAT	X	X	ok	ok	O
FLOAT	X	X	F	ok	O
DECIMAL	X	X	F	F	O

Legend:

ok = no error

O = error can occur depending on precision of the decimal

X = error can occur depending on data

F = no error, but less significant digits can be lost

For example, if you convert a FLOAT column to DECIMAL(4,2), your database server rounds off the floating-point numbers before storing them as decimal numbers. This conversion can result in an error depending on the precision assigned to the DECIMAL column.

Converting Between Number and CHAR

You can convert a CHAR column to a number column and vice versa. However, if the CHAR column contains any characters that are not valid in a number column (for example, the letter *l* instead of the number *1*), your database server is unable to complete the ALTER TABLE statement and leaves the column values as characters.

Converting Between DATE and DATETIME

You can convert DATE columns to DATETIME columns. However, if the DATETIME column contains more fields than the DATE column, the database server either ignores the fields or fills them with zeros. The following examples illustrate how these two data types are converted (assuming that the default date format is *mm/dd/yyyy*):

- If you convert DATE to DATETIME YEAR TO DAY, the database server converts the existing DATE values to DATETIME values. For example, the value 08/15/1990 becomes 1990-08-15.
- If you convert DATETIME YEAR TO DAY to DATE, the value 1990-08-15 becomes 08/15/1990.
- If you convert DATE to DATETIME YEAR TO SECOND, the database server converts existing DATE values to DATETIME values and fills in the additional DATETIME fields with zeros. For example, 08/15/1990 becomes 1990-08-15 00:00:00.
- If you convert DATETIME YEAR TO SECOND to DATE, the database server converts existing DATETIME to DATE values but drops fields more precise than DAY. For example, 1990-08-15 12:15:37 becomes 08/15/1990.

Range of Operations Using DATE, DATETIME, and INTERVAL

You can use DATE, DATETIME, and INTERVAL data in a variety of arithmetic and relational expressions. You can manipulate a DATETIME value with another DATETIME value, an INTERVAL value, the current time (identified by the keyword CURRENT), or a specified unit of time (identified by the keyword UNITS). In most situations, you can use a DATE value wherever it is appropriate to use a DATETIME value and vice versa. You also can manipulate an INTERVAL value with the same choices as a DATETIME value. In addition, you can multiply or divide an INTERVAL value by a number.

An INTERVAL column can hold a value that represents the difference between two DATETIME values or the difference between (or sum of) two INTERVAL values. In either case, the result is a span of time, which is an INTERVAL value. On the other hand, if you add or subtract an INTERVAL value from a DATETIME value, another DATETIME value is produced because the result is a specific point in time.

Figure 3-8 indicates the range of expressions that you can use with DATE, DATETIME, and INTERVAL data, along with the data type that results from each expression.

Figure 3-8
Range of expression for DATE, DATETIME, and INTERVAL

Data Type of Operand 1	Operator	Data Type of Operand 2	Result
DATE	-	DATETIME	INTERVAL
DATETIME	-	DATE	INTERVAL
DATE	+ or -	INTERVAL	DATETIME
DATETIME	-	DATETIME	INTERVAL
DATETIME	+ or -	INTERVAL	DATETIME
INTERVAL	+	DATETIME	DATETIME
INTERVAL	+ or -	INTERVAL	INTERVAL

(1 of 2)

Data Type of Operand 1	Operator	Data Type of Operand 2	Result
DATETIME	-	CURRENT	INTERVAL
CURRENT	-	DATETIME	INTERVAL
INTERVAL	+	CURRENT	DATETIME
CURRENT	+ or -	INTERVAL	DATETIME
DATETIME	+ or -	UNITS	DATETIME
INTERVAL	+ or -	UNITS	INTERVAL
INTERVAL	* or /	NUMBER	INTERVAL

(2 of 2)

No other combinations are allowed. You cannot add two DATETIME values because this operation does not produce either a point in time or a span of time. For example, you cannot add December 25 and January 1, but you can subtract one from the other to find the time span between them.

Manipulating DATETIME Values

You can subtract most DATETIME values from each other. Dates can be in any order and the result is either a positive or a negative INTERVAL value. The first DATETIME value determines the field precision of the result.

If the second DATETIME value has fewer fields than the first, the shorter value is extended automatically to match the longer one. (See the EXTEND syntax explanation on page [page 7-387](#).) In the following example, subtracting the DATETIME YEAR TO HOUR value from the DATETIME YEAR TO MINUTE value results in a positive interval value of 60 days, 1 hour, and 30 minutes. Since minutes were not included in the second value, the database server sets the minutes for the result to 0.

```
DATETIME (1990-9-30 12:30) YEAR TO MINUTE
- DATETIME (1990-8-1 11) YEAR TO HOUR

Result: INTERVAL (60 01:30) DAY TO MINUTE
```

If the second DATETIME value has more fields than the first (regardless of whether the precision of the extra fields is larger or smaller than those in the first value), the additional fields in the second value are ignored in the calculation. In the following expression (and result), the year is not included for the second value. Therefore, the year is set automatically to the current year, in this case 1990, and the resulting INTERVAL is negative, indicating that the second date is later than the first.

```
DATETIME (1990-9-30) YEAR TO DAY
- DATETIME (10-1) MONTH TO DAY

Result: INTERVAL (1) DAY TO DAY [assuming current year is 1990]
```

Manipulating DATETIME with INTERVAL Values

INTERVAL values can be added to or subtracted from DATETIME values. In either case, the result is a DATETIME value. If you are adding an INTERVAL value to a DATETIME value, the order of values is unimportant; however, if you are subtracting, the DATETIME value must come first. Adding or subtracting an INTERVAL value simply moves the DATETIME value forward or backward in time. In the following example, the expression moves the date ahead three years and five months:

```
DATETIME (1990-8-1) YEAR TO DAY
+ INTERVAL (3-5) YEAR TO MONTH

Result: DATETIME (1994-01-01) YEAR TO DAY
```

In most situations, the database server automatically adjusts the calculation when the initial values do not have the same precision. However, in certain situations you must explicitly adjust the precision of one value to perform the calculation. If the INTERVAL value you are adding or subtracting has fields that are not included in the DATETIME value, you must use the EXTEND function to explicitly extend the field qualifier of the DATETIME value. (For more information on the EXTEND function, see [“EXTEND Function” on page 7-387](#).) For example, you cannot subtract a minute INTERVAL value from the DATETIME value in the previous example that has a YEAR TO DAY field qualifier. You can, however, use the EXTEND function to perform this calculation, as shown in the following example:

```
EXTEND (DATETIME (1990-8-1) YEAR TO DAY, YEAR TO MINUTE)
- INTERVAL (720) MINUTE(3) TO MINUTE

Result: DATETIME (1990-07-31 12:00) YEAR TO MINUTE
```

The EXTEND function allows you to increase explicitly the DATETIME precision from YEAR TO DAY to YEAR TO MINUTE. This allows the database server to perform the calculation, with the resulting extended precision of YEAR TO MINUTE.

Manipulating DATE with DATETIME and INTERVAL Values

You can use DATE values in arithmetic expressions with DATETIME or INTERVAL values by writing expressions that allow the manipulations shown in [Figure 3-9](#).

Figure 3-9

Results of expressions that manipulate DATE with DATETIME or INTERVAL values

Expression	Result
DATE - DATETIME	INTERVAL
DATETIME - DATE	INTERVAL
DATE + or - INTERVAL	DATETIME

In these cases, DATE values are first converted to their corresponding DATETIME equivalents, and then the expression is computed in the normal way.

While you can interchange DATE and DATETIME values in many situations, you must indicate whether a value is a DATE or a DATETIME data type. A DATE value can come from any of the following sources:

- A column or program variable of type DATE
- The TODAY keyword
- The DATE() function
- The MDY function

A DATETIME value can come from any of the following sources:

- A column or program variable of type DATETIME
- The CURRENT keyword
- The EXTEND function
- A DATETIME literal

When you represent DATE and DATETIME values as quoted character strings, the fields in the strings must be in proper order. In other words, when a DATE value is expected, the string must be in DATE format and when a DATETIME value is expected, the string must be in DATETIME format. For example, you can use the string "10/30/1990" as a DATE string but not as a DATETIME string. Instead, you must use "1990-10-30" or "90-10-30" as the DATETIME string.

You also can subtract one DATE value from another DATE value, but the result is a positive or negative INTEGER value, rather than an INTERVAL value. If an INTERVAL value is required, you either can convert the INTEGER value into an INTERVAL value or one of the DATE values into a DATETIME value before subtracting.

For example, the following expression uses the DATE() function to convert character string constants to DATE values, calculates their difference, and then uses the UNITS DAY keywords to convert the INTEGER result into an INTERVAL value:

```
(DATE ("5/2/1990") - DATE ("4/6/1955")) UNITS DAY
```

```
Result: INTERVAL (12810) DAY(5) TO DAY
```

If you need YEAR TO MONTH precision, you can use the EXTEND function on the first DATE operand, as illustrated in the next example:

```
EXTEND (DATE ("5/2/1990"), YEAR TO MONTH) - DATE ("4/6/1955")
```

```
Result: INTERVAL (35-01) YEAR TO MONTH
```

Note that the resulting INTERVAL precision is YEAR TO MONTH because the DATETIME value came first. If the DATE value had come first, the resulting INTERVAL precision would have been DAY(5) TO DAY.

Manipulating INTERVAL Values

You can add INTERVAL values together or subtract them from each other as long as both values are from the same class; that is, both are year-month or both are day-time. In the following example, a SECOND TO FRACTION value is subtracted from a MINUTE TO FRACTION value:

```
INTERVAL (100:30.0005) MINUTE(3) TO FRACTION(4)
- INTERVAL (120.01) SECOND(3) TO FRACTION
```

```
Result: INTERVAL (98:29.9905) MINUTE TO FRACTION(4)
```

Note the use of numeric qualifiers to alert the database server that the MINUTE and FRACTION in the first value, and the SECOND in the second value, exceed the default number of digits.

When you add or subtract INTERVAL values, the second value cannot have a field with greater precision than the first. For example, the second INTERVAL cannot be YEAR TO MONTH if the first is MONTH TO MONTH. The second INTERVAL, however, can have a field of smaller precision than the first. For example, the second INTERVAL can be HOUR TO SECOND when the first is DAY TO HOUR. The additional fields (in this case MINUTE and SECOND) in the second INTERVAL value are ignored in the calculation.

Multiplying or Dividing INTERVAL Values

You can multiply or divide INTERVAL values by a number. The number can be an integer or a fraction. However, if there is a remainder from the calculation, it is ignored, and the result is truncated. For example, the following expression multiplies an INTERVAL by a fraction:

```
INTERVAL (15:30.0002) MINUTE TO FRACTION(4) * 2.5
```

```
Result: INTERVAL (38:45.0005) MINUTE TO FRACTION(4)
```

In this example, $15 * 2.5 = 37.5$ minutes, $30 * 2.5 = 75$ seconds, and $2 * 2.5 = 5$ fraction(4). The .5 minute is converted into 30 seconds and 60 seconds are converted into 1 minute, which produces the final result of 38 minutes, 45 seconds, and .0005 of a second. Note that the results of any calculation include the same amount of precision as the original INTERVAL value.

Environment Variables

In This Chapter	4-3
Setting Environment Variables	4-4
Informix Environment Variables	4-5
DBANSIWARN.	4-7
DBDATE	4-8
DBDELIMITER.	4-9
DBEDIT	4-10
DBFORMAT.	4-10
DBLANG.	4-11
DBMENU.	4-12
DBMONEY	4-12
DBNETTYPE	4-13
DBPATH	4-14
DBPRINT.	4-15
DBREMOTECMD.	4-15
DBSRC.	4-16
DBTEMP	4-17
DBTIME	4-17
INFORMIXCOB	4-20
INFORMIXCOBDIR	4-20
INFORMIXCOBSTORE	4-21
INFORMIXCOBTYPE	4-21
INFORMIXDIR.	4-22
INFORMIXONLINEDIR	4-23
INFORMIXTERM	4-23
NOSORTINDEX	4-24
SQLEXEC.	4-25

SQLRM	4-26
SQLRMDIR	4-27
TBCONFIG	4-27
UNIX Environment Variables	4-28
PATH	4-28
TERM	4-29
TERMCAP	4-29
TERMINFO	4-30

In This Chapter

To use your IBM Informix products, you must set certain environment variables that identify your terminal, specify the location of your software, and define other parameters of the environment of your product. This chapter describes all the environment variables that apply to one or more IBM Informix products and tells how to set them. It also shows how to set certain UNIX system environment variables.

Your IBM Informix product makes the following default assumptions about your environment:

- Temporary files are stored in the **/tmp** directory.
- The program, compiler, or preprocessor of your product, and any associated files and libraries, have been installed in the **/usr/informix** directory. For specifics, see your product manual.
- If you are using IBM Informix SE, the target or current database is in the current directory.
- The field separator for unloaded data files is the vertical bar (`|=ASCII 124`).
- The editor for products that use one is the predominant editor for the operating system, usually **vi**.
- For products that have a print capability, if the computer is running UNIX System V, the program that sends files to the printer is usually **lp**. For other UNIX systems, the default is **lpr**.

Setting Environment Variables

You can change any of the preceding defaults by setting one or more of the environment variables recognized by your IBM Informix product. If you are already using an IBM Informix product, some or all of the appropriate environment variables might already be set. Enter `printenv` (BSD UNIX) or `env` (UNIX System V) at the system prompt to view your current environment settings.

You can set Informix and UNIX environment variables at the system prompt or in your `.login` or `.cshrc` (C shell) or your `.profile` (Bourne shell) file. (Make sure that an environment variable is not set differently in your `.login` and `.cshrc` files.)

- When you set an environment variable at the system prompt, you must reassign it the next time you log into the system.
- When you set an environment variable in your `.login`, `.cshrc`, or `.profile` file, it is assigned automatically every time you log into the system.

If you set the variables in a file, you should log out and then log back in, or `source` the file before you work with your IBM Informix product, to allow the shell to read your entries.

You use standard UNIX commands to set environment variables. For example, you can set the `ABCD` environment variable to *value* as follows:

C shell: `setenv ABCD value`

Bourne shell: `ABCD=value`
 `export ABCD`

The figure that follows lists all the environment variables specific to IBM Informix products. These environment variables and their uses are discussed in the next section. The standard UNIX system environment variables `PATH`, `TERM`, `TERMCAP`, and `TERMINFO` are described in the last section of this chapter.

In addition, the IBM Informix OnLine administrator can elect to set additional environment variables to preserve diagnostic information and for parallel sort capability. The four consistency checking variables are **GCORE**, **DUMPCORE**, **DUMPDIR**, and **DUMPSHMEM**; the two parallel sort variables are **PSORT_NPROCS** and **PSORT_DBTEMP**. Their use and setting are discussed in *IBM Informix OnLine Administrator's Guide*.

Informix Environment Variables

[Figure 4-1](#) lists alphabetically the environment variables that IBM Informix products support.

Figure 4-1
Environment variables in IBM Informix products

	ISQL	DB-Access	I4GL	Interactive Debugger	ESQL/C	ESQL/ COBOL
DBANSIWARN	■	■	■	■	■	■
DBDATE	■	■	■	■	■	■
DBDELIMITER	■	■	■	■		
DBEDIT	■	■	■	■		
DBFORMAT	■		■			
DBLANG	■	■	■	■	■	■
DBMENU	■					
DBMONEY	■	■	■	■	■	■
DBNETTYPE	■	■	■	■	■	■
DBPATH	■	■	■	■	■	■
DBPRINT	■	■	■	■	■	
DBREMOTECMD	■	■	■	■	■	■
DBSCREENOUT						
DBSRC				■		
DBTEMP	■	■	■	■	■	■
DBTIME					■	■
INFORMIXCOB						■
INFORMIXCOBDIR						■
INFORMIXCOBSTORE						■
INFORMIXCOBTYPE						■
INFORMIXDIR	■	■	■	■	■	■
INFORMIXONLINEDIR	■	■	■		■	■
INFORMIXTERM	■	■	■			
NOSORTINDEX	■	■	■	■	■	■
SQLEXEC	■	■	■	■	■	■
SQLRM					■	■
SQLRMDIR					■	■
TBCONFIG	■	■	■	■	■	■

DBANSIWARN

The **DBANSIWARN** environment variable indicates that you want to check for Informix extensions. Unlike most environment variables, you do not need to set **DBANSIWARN** to a value. Setting it to any value or to no value, as follows, is sufficient:

C shell: `setenv DBANSIWARN`

Bourne shell: `DBANSIWARN=
export DBANSIWARN`

Setting the **DBANSIWARN** environment variable for IBM Informix SQL is functionally equivalent to invoking **isql** or **saceprep** with the **-ansi** flag. (For DB-Access, it is equivalent to invoking the utility from the command line with the **-ansi** flag.) If you set **DBANSIWARN** before you run **isql** or **saceprep**, warnings are generated when Informix extensions to ANSI standard syntax are encountered. Within the SQL menu, warnings are displayed on the screen. From **saceprep** (or within the **REPORT** menu), warnings are written to a **.err** file.

Setting the **DBANSIWARN** environment variable before you compile an IBM Informix 4GL or an IBM Informix ESQL program is functionally equivalent to specifying the **-ansi** flag in the command line. When Informix extensions to ANSI standard syntax are encountered in your program, warning messages are written to the screen for the IBM Informix ESQL products or to an **.err** file for IBM Informix 4GL.

At run time, the **DBANSIWARN** environment variable causes the following SQL Communication Area (**SQLCA**) variable to be set to **W** when a statement that is not ANSI-compliant is executed. (For more information on **SQLCA**, see [Chapter 5, "Error Handling with SQLCA."](#))

Product	SQLCA Variable
IBM Informix ESQL/C	sqlca.sqlwarn.sqlwarn5
IBM Informix ESQL/COBOL	SQLWARN5 OF SQLWARN OF SQLCA
IBM Informix 4GL	SQLCA.SQLAWARN[6]

Once you set **DBANSIWARN**, Informix extension checking is automatic until you log out or reset **DBANSIWARN**. To turn off Informix extension checking, reset the **DBANSIWARN** environment variable with the following command:

C shell: `unsetenv DBANSIWARN`

Bourne shell: `unset DBANSIWARN`

DBDATE

The **DBDATE** environment variable specifies the following formats for DATE values:

- The order of the month, day, and year in a date
- Whether the year should be printed with two digits (Y2) or four digits (Y4)
- The separator between the month, day, and year

The default value for **DBDATE** is **MDY4/**, where **M** represents the month, **D** represents the day, **Y4** represents a four-digit year, and the slash (/) is a separator. For example: 12/25/1990.

Other acceptable characters for the separator are a hyphen (-), a period (.), or a zero (0). (Use the zero to indicate no separator.)

The slash (/) appears if you attempt to use a character other than a hyphen, period, or zero as a separator, or if you do not include a separator character in the **DBDATE** definition.

You always must specify the separator character last. The number of digits you specify for the year must always follow the **y**.

To make the date appear as *mmddy*, set the **DBDATE** environment variable as follows:

C shell: `setenv DBDATE MDY20`

Bourne shell: `DBDATE=MDY20
export DBDATE`

Here, **MDY** represents the order of month, day, and year; 2 indicates two digits for the year; and 0 specifies no separator. As a result, the date is displayed as 122590.

To make the date appear in European format (*dd-mm-yyyy*), set the **DBDATE** environment variable as follows:

C shell: `setenv DBDATE DMY4-`

Bourne shell: `DBDATE=DMY4-`
`export DBDATE`

Here, **DMY** represents the order of day, month, and year; 4 indicates four digits for the year; and - specifies a hyphen separator. As a result, the date is displayed as 25-12-1990.



Tip: *Certain routines called by ESQL/C and ESQL/COBOL can use the **DBTIME** variable, rather than **DBDATE**, to set DATETIME formats to international specifications. See the description of **DBTIME** and your ESQL product manual for more information.*

DBDELIMITER

The **DBDELIMITER** environment variable specifies the field delimiter used by the **dbload** and **dbexport** utilities in unloaded data files or with the **LOAD** and **UNLOAD** statements in IBM Informix 4GL, IBM Informix SQL, and DB-Access. The vertical bar (`|=ASCII 124`) is the default.

For example, to change the field delimiter to a plus (+), set the **DBDELIMITER** environment variable as follows:

C shell: `setenv DBDELIMITER +`

Bourne shell: `DBDELIMITER=+`
`export DBDELIMITER`

DBEDIT

The **DBEDIT** environment variable names the text editor you want to use for the following tasks:

- To modify form specification files, report specification files, and command files from within IBM Informix SQL
- To work with SQL statements and command files in DB-Access
- To create program files, form specification files, and command files from within the IBM Informix 4GL Programmer Environment

If **DBEDIT** is set, the specified editor is called directly. If **DBEDIT** is not set, you are prompted to specify an editor as the default for the rest of the session.

Set the **DBEDIT** environment variable as follows:

C shell: `setenv DBEDIT editor`

Bourne shell: `DBEDIT=editor`
`export DBEDIT`

where *editor* is the name of the text editor you want to use.

For most systems, the default is **vi**. If you use another editor, be sure that it is installed to create flat ASCII files. Some word processors in “document mode” introduce printer control characters that can interfere with operation of IBM Informix 4GL, IBM Informix 4GL Interactive Debugger, IBM Informix SQL ACE report writer, or IBM Informix SQL PERFORM screen transaction processor.

DBFORMAT

The **DBFORMAT** environment variable specifies the format that IBM Informix SQL and IBM Informix 4GL use in the following situations:

- When you enter values on a screen
- When you use the IBM Informix SQL ACE report writer or PERFORM screen transaction processor
- When IBM Informix 4GL displays output values on a screen or in a report

This environment variable is used to format values of the DECIMAL, MONEY, FLOAT, SMALLFLOAT, INTEGER, and SMALLINT data types. The formatting specified by the **DBFORMAT** variable overrides any formatting specified by the **DBMONEY** variable, except in the character string generated by the 4GL CONSTRUCT statement. (See your product manual for details.)

Set the **DBFORMAT** environment variable as follows:

C shell: `setenv DBFORMAT 'front:thousands:decimal:back'`

Bourne shell: `DBFORMAT='front:thousands:decimal:back'`
`export DBFORMAT`

Here, *front* is an optional one- to seven-character value for the leading currency symbol. The optional *thousands* is one or more characters that indicate the possible thousands separator. It can be anything except digits, <, >, |, :, ?, !, =, [, or]; the default is *. The *decimal* is one or more characters that determine the possible decimal separators. It can be anything except digits, <, >, |, :, ?, !, =, [,], *, or any characters specified for the *thousands* value. The *back* represents an optional one- to seven-character value for the trailing currency symbol.

DBLANG

The **DBLANG** environment variable specifies the subdirectory of **\$INFORMIXDIR** that contains the compiled message files used by your program. The default subdirectory is **msg** and the compiled message files have the suffix **.iem**.

If you want to use a message directory other than **\$INFORMIXDIR/msg**, follow these steps:

1. Use the **mkdir** command to create the appropriate subdirectory in **\$INFORMIXDIR**.
2. Set the owner and group of the subdirectory to **informix** and the access permission for this directory to **755**.

3. Set the **DBLANG** environment variable to the new subdirectory as follows (specify only the name of the subdirectory and not its pathname):

C shell: `setenv DBLANG dirname`

Bourne shell: `DBLANG=dirname`
`export DBLANG`
4. Copy the **.iem** files to the new message directory specified by **\$INFORMIXDIR/\$DBLANG**. All files in the message directory should have the owner and group **informix** and access permission **644**.
5. Run your program or otherwise continue working with your product.

DBMENU

The **DBMENU** environment variable specifies the user-menu that IBM Informix SQL accesses first. The default is the Main Menu of the user-menu for the current database. For example, to make the “testing” menu the default, set the **DBMENU** environment variables as follows:

C shell: `setenv DBMENU testing`

Bourne shell: `DBMENU=testing`
`export DBMENU`

DBMONEY

The **DBMONEY** environment variable specifies the display format for MONEY values and consists of [*front*][. | ,][*back*].

Here, *front* is the optional symbol that precedes the MONEY value. The period or comma is the optional symbol that separates the integral from the fractional part of the MONEY value. The *back* represents the optional symbol that follows the MONEY value. The *front* and *back* symbols can be up to seven characters long and can contain any character except a comma or period.

The default value for **DBMONEY** is \$., where a dollar sign precedes the MONEY value, a period (.) separates the integral from the fractional part of the MONEY value, and no back symbol appears. For example: \$100.50.

Suppose you want to represent MONEY values in *DM* (*Deutsche Mark*), which uses the currency symbol **DM** and a comma. Set the **DBMONEY** environment variable as follows:

C shell: `setenv DBMONEY DM,`

Bourne shell: `DBMONEY=DM,
export DBMONEY`

Here, **DM** is the currency symbol preceding the MONEY value, and a comma separates the integral from the fractional part of the MONEY value. As a result, the amount is displayed as **DM100,50**.

Whenever you make a change to the *back* symbol, you must also supply the *front* symbol and the MONEY value separator (comma or period). Similarly, if you change the value separator from a comma to a period, you must also supply the *front* symbol.

DBMONEY specifies the display format for the data types MONEY, DECIMAL, and FLOAT.



Tip: *IBM Informix 4GL uses the **DBFORMAT** variable, rather than **DBMONEY**, to format certain data type values displayed on a screen or in a report. See the description of **DBFORMAT** in this chapter and the product manual for more information about how IBM Informix 4GL formats output values.*

DBNETTYPE

DBNETTYPE is an environment variable specific to certain AT&T platforms and is used to optimize the identification of a network protocol. You set **DBNETTYPE** only if you are using an AT&T machine with IBM Informix NET or IBM Informix STAR, have both a **/dev/starlan** and either a **/dev/tcp** or a **/dev/it** file on your client and server machines, and want to use **tcp/ip** instead of **starlan**. Set **DBNETTYPE** on the server before invoking the **sqlxecd** daemon.

If **DBNETTYPE** is not set, the default is StarLAN. Specify TCP/IP as follows:

C shell: `setenv DBNETTYPE tcp/ip`

Bourne shell: `DBNETTYPE=tcp/ip`
`export DBNETTYPE`

For more information on network protocols, see the *IBM Informix NET and IBM Informix STAR Installation and Configuration Guide*.

DBPATH

The **DBPATH** environment variable specifies a list of directories (in addition to the current directory) for your product to search for reports, forms, query files, command scripts, and so on. If you are using the IBM Informix SE database server, **DBPATH** also identifies directories that contain databases. If you are using IBM Informix NET or IBM Informix STAR, **DBPATH** also can include remote host names and paths. See the *IBM Informix NET and IBM Informix STAR Installation and Configuration Guide* for details.

Use the same format that you use to set the **PATH** variable. Make sure you enter a colon between the directory names. For example, the following **DBPATH** setting causes your product to search for database files in Nigel's and Zooie's directories as well as in your current directory:

C shell: `setenv DBPATH /usr/Nigel:/usr/Zooie`

Bourne shell: `DBPATH=/usr/Nigel:/usr/Zooie`
`export DBPATH`

In an IBM Informix NET or IBM Informix STAR environment, you can set **DBPATH** to directories on either local or remote machines by including a double slash (//) in front of the machine name.

For example, this setting specifies the **/results** directory on the **quality** machine:

C shell: `setenv DBPATH //quality/results`

Bourne shell: `DBPATH=//quality/results`
`export DBPATH`

The following example sets **DBPATH** to access remote IBM Informix OnLine databases on the **prodmar** machine:

C shell: `setenv DBPATH //prodmar`

Bourne shell: `DBPATH=//prodmar`
`export DBPATH`



Tip: The IBM Informix 4GL Interactive Debugger uses the **DBSRC** variable, rather than **DBPATH**, to search for 4GL program source files. See the description of **DBSRC** in this chapter and your product manual for more information about how the IBM Informix 4GL Interactive Debugger searches for 4GL source files.

DBPRINT

The **DBPRINT** environment variable specifies the print program for your computer. You can name any command, shell script, or UNIX utility that handles standard ASCII input. For most BSD UNIX systems, the default program is **lpr**. For UNIX System V, the default program is usually **lp**.

Set the **DBPRINT** environment variable as follows:

C shell: `setenv DBPRINT progrname`

Bourne shell: `DBPRINT=progrname`
`export DBPRINT`

where *progrname* is the name of the print program you want to use.

DBREMOTECMD

You can set the **DBREMOTECMD** environment variable to override the default remote shell specified for your implementation of IBM Informix OnLine. Set it using either a simple command or the full pathname. If you use the full pathname, the database server searches your **PATH** for the specified command.

The full pathname syntax is highly recommended on the interactive UNIX platform to avoid problems with like-named programs in other directories and possible confusion with the “restricted shell” (**/usr/bin/rsh**).

Set the **DBREMOTECMD** environment variable as follows for a simple command name:

C shell: `setenv DBREMOTECMD rcmd`

Bourne shell: `DBREMOTECMD=rcmd`
`export DBREMOTECMD`

Set the **DBREMOTECMD** environment variable as follows to specify the full pathname:

C shell: `setenv DBREMOTECMD /usr/bin/remsh`

Bourne shell: `DBREMOTECMD=/usr/bin/remsh`
`export DBREMOTECMD`

For more information, see the discussion of the remote tape facility for IBM Informix OnLine archives, restores, and logical log backups in *IBM Informix OnLine Administrator's Guide*.

DBSRC

DBSRC is an environment variable specific to the IBM Informix 4GL Interactive Debugger. It specifies directory pathnames that are part of the search path only during debugging sessions. Make sure you enter a colon between the directory names.

For example, the following **DBSRC** setting causes the IBM Informix 4GL Interactive Debugger to search for files in the **programs** and **june** directories:

C shell: `setenv DBSRC /b/shawn/programs:/b/june`

Bourne shell: `DBSRC=/b/shawn/programs:/b/june`
`export DBSRC`

If you do not specify a **DBSRC** variable, the current directory is the default.

When you exit from the IBM Informix 4GL Interactive Debugger and return to the 4GL Programmer Environment or to the operating system, the pathnames specified in **DBSRC** are no longer part of the search path.

See your *Guide to the IBM Informix 4GL Interactive Debugger* for more information on specifying the order of directory search during debugging sessions.

DBTEMP

The **DBTEMP** environment variable specifies the directory into which your product places its temporary files. You need not set **DBTEMP** if the default, **/tmp**, is satisfactory. Set the **DBTEMP** environment variable as follows:

C shell: `setenv DBTEMP dirname`

Bourne shell: `DBTEMP=dirname`
`export DBTEMP`

where *dirname* is the full pathname of the directory you want to hold temporary files.

DBTIME

The **DBTIME** environment variable can be set to allow you to manipulate DATETIME formats so they conform more closely to various international or local TIME conventions. **DBTIME** takes effect only when you call certain IBM Informix ESQL DATETIME routines; otherwise, you should use the **DBDATE** environment variable. (See your *IBM Informix ESQL/C Programmer's Manual* or *IBM Informix ESQL/COBOL Programmer's Manual* for details.)

You can set **DBTIME** to specify the exact format of an input/output (I/O) DATETIME string field by using the following formatting directives. Otherwise, the behavior of the DATETIME formatting routine is undefined.

String	Use
%b	is replaced by the abbreviated month name.
%B	is replaced by the full month name.
%d	is replaced by the day of the month as a decimal number [01,31].

(1 of 2)

String	Use
%Fn	is replaced by the value of the fraction with precision specified by the integer <i>n</i> . The default value of <i>n</i> is 2; the range of <i>n</i> is $0 \leq n \leq 5$.
%H	is replaced by the hour (24-hour clock) as a decimal number [00,23].
%I	is replaced by the hour (12-hour clock) as a decimal number [01,12].
%M	is replaced by the minute as a decimal number [00,59].
%m	is replaced by the month as a decimal number [01,12].
%p	is replaced by "a.m." or "p.m." (or the equivalent in the local standards).
%S	is replaced by the second as a decimal number [00,59].
%y	is replaced by the year as a 2-digit decimal number [00,99]. The format for an interval value is taken literally: "88" means "0088", not "1988."
%Y	is replaced by the year as a 4-digit decimal number; use Y for an interval of more than 99 years.
%%	is replaced by % (to allow % in the format string).

(2 of 2)

For example, to convert a *DATETIME year to second* to an ASCII string format that looks like this:

Mar 21, 1990 at 16 h 30 m 28 s

you set **DBTIME** as follows:

C shell: `setenv DBTIME "%b %d, %Y at %H h %M m %S s"`

Bourne shell: `DBTIME="%b %d, %Y at %H h %M m %S s"`
`export DBTIME`

The default **DBTIME**, which produces the conventional ANSI SQL string format that looks like this:

1990-03-21 16:30:28

is set as follows:

C shell: `setenv DBTIME "%Y-%m-%d %H:%M:%S"`

Bourne shell: `DBTIME="%Y-%m-%d %H:%M:%S"`
`export DBTIME`

An optional field width and precision specification can immediately follow the % character; it is interpreted as follows:

`[-|0]w` where *w* is a decimal digit string specifying the minimum field width. By default, the value is right-justified with spaces on the left.

If `-` is specified, it is left-justified with spaces on the right.

If `0` is specified, it is right-justified and padded with zeroes on the left.

`.p` where *p* is a decimal digit string specifying the number of digits to appear for `d`, `H`, `I`, `m`, `M`, `S`, `y`, and `Y` conversions, and the maximum number of characters to be used for `b` and `B` conversions. A precision specification is significant only when converting a DATETIME value to an ASCII string and not vice versa.

- If a conversion specification supplies fewer digits than specified by a precision, it is padded with leading zeroes.
- If a conversion specification supplies more characters than specified by a precision, excess characters are truncated on the right.
- If no field width or precision is specified for `d`, `H`, `I`, `m`, `M`, `S`, or `y` conversions, a default of `.2` is used. A default of `.4` is used for `Y` conversions.

The `F` conversion does not follow the field width and precision format conversions described earlier.

INFORMIXCOB

The **INFORMIXCOB** environment variable specifies the program name of the COBOL compiler you use with IBM Informix ESQL/COBOL. (Refer to your COBOL system manual for the name of your COBOL compiler.) Set the **INFORMIXCOB** environment variable as follows:

C shell: `setenv INFORMIXCOB program`

Bourne shell: `INFORMIXCOB=program`
`export INFORMIXCOB`

where *program* is the program name of the COBOL compiler, that is, the command that calls up the compiler environment.

INFORMIXCOBDIR

The **INFORMIXCOBDIR** environment variable is the directory where the run-time library and objects reside. This environment variable is used only when you create a COBOL run-time program with IBM Informix ESQL/COBOL.

Set the **INFORMIXCOBDIR** environment variable as follows:

C shell: `setenv INFORMIXCOBDIR dirname`

Bourne shell: `INFORMIXCOBDIR=dirname`
`export INFORMIXCOBDIR`

where *dirname* is the name of the directory you want.

INFORMIXCOBSTORE

You can set the **INFORMIXCOBSTORE** environment variable to indicate to ESQL/COBOL the type of storage for IBM Informix ESQL/COBOL to use during compilation in the MF COBOL/2 environment. This variable enables ESQL/COBOL to allow or disallow certain PICTURE clauses that are mapped to internal C variable types.

The number of bytes needed to store BINARY or COMPUTATIONAL data is based on the size (maximum number of digits) specified in the PICTURE clause. The MF COBOL/2 compiler also considers whether *byte* or *word* storage is specified when determining the number of bytes needed to store BINARY and COMPUTATIONAL data. (MF COBOL/2 uses only byte storage.)

If left undefined, the default storage mode is byte, which is more restrictive of available data type choices. If you are using byte storage, the only legal PIC sizes are 3, 4, 7, 8, and 9. If you are using word storage, PIC sizes can range from 1 through 9.

To specify word storage, set the **INFORMIXCOBSTORE** environment variable as follows:

C shell: `setenv INFORMIXCOBSTORE word`

Bourne shell: `INFORMIXCOBSTORE=word`
`export INFORMIXCOBSTORE`

For a table showing the storage allocation for Micro Focus compilers, see your *IBM Informix ESQL/COBOL Programmer's Manual*.

INFORMIXCOBTYPE

The **INFORMIXCOBTYPE** environment variable is a two-character code that specifies the manufacturer of your COBOL compiler. Set the **INFORMIXCOBTYPE** environment variable as follows:

C shell: `setenv INFORMIXCOBTYPE type`

Bourne shell: `INFORMIXCOBTYPE=type`
`export INFORMIXCOBTYPE`

where *type* is the manufacturer of the COBOL compiler that you use with IBM Informix ESQL/COBOL. For example:

Type	Manufacturer
mf2	Micro Focus
rm85	Ryan-McFarland

See your *IBM Informix ESQL/COBOL Programmer's Manual* for compiler-specific information.

INFORMIXDIR

The **INFORMIXDIR** environment variable specifies the directory that contains the subdirectories in which your product files are installed. If you have multiple versions of IBM Informix OnLine or IBM Informix SE, set **INFORMIXDIR** to the appropriate directory name for the version that you want to access.

Set the **INFORMIXDIR** environment variable to the following recommended installation directory:

C shell: `setenv INFORMIXDIR /usr/informix`

Bourne shell: `INFORMIXDIR=/usr/informix`
 `export INFORMIXDIR`

INFORMIXONLINEDIR

The **INFORMIXONLINEDIR** environment variable specifies the directory in which your IBM Informix OnLine server is installed. If you have multiple versions of IBM Informix OnLine, set **INFORMIXONLINEDIR** to the appropriate directory name for the version that you want to access.

Set the **INFORMIXONLINEDIR** environment variable to the following recommended installation directory:

C shell: `setenv INFORMIXONLINEDIR /usr/informix`

Bourne shell: `INFORMIXONLINEDIR=/usr/informix`
`export INFORMIXONLINEDIR`

See the *IBM Informix OnLine Administrator's Guide* for information on when to use this variable.

INFORMIXTERM

The **INFORMIXTERM** environment variable specifies whether IBM Informix SQL, IBM Informix 4GL, and DB-Access should use the information in the **termcap** file or the **terminfo** directory. **INFORMIXTERM** determines terminal-dependent keyboard and screen capabilities such as the operation of function keys, color and intensity attributes in screen displays, and the definition of window border and graphics characters.

If **INFORMIXTERM** is not set, the default is **termcap**. When IBM Informix SQL, IBM Informix 4GL, or DB-Access is installed on your system, a **termcap** file is placed in the **etc** subdirectory of **\$INFORMIXDIR**. This file is a superset of an operating system **termcap** file.

You can use the **termcap** file supplied by Informix, the system **termcap** file, or a **termcap** file that you created yourself. You must set the **TERMCAP** environment variable if you do not use the default **termcap** file.

The **terminfo** directory contains a file for each terminal name that has been defined. It is supported only on machines that provide full support for the UNIX System V **terminfo** library.

The entry for your terminal might allow you to use the REVERSE and UNDERLINE intensity attributes. However, you must set **INFORMIXTERM** to **termcap** if you use color or the intensity attributes BLINK or BOLD in IBM Informix SQL screen forms, or if you use color or the BLINK, BOLD, DIM, or INVISIBLE attributes in IBM Informix 4GL programs and screen displays.

Set the **INFORMIXTERM** environment variable to **terminfo** as follows:

C shell: `setenv INFORMIXTERM terminfo`

Bourne shell: `INFORMIXTERM=terminfo`
 `export INFORMIXTERM`

Set the **INFORMIXTERM** environment variable to **termcap** as follows.

C shell: `setenv INFORMIXTERM termcap`

Bourne shell: `INFORMIXTERM=termcap`
 `export INFORMIXTERM`



Tip: If **INFORMIXTERM** is set to **termcap**, you must set the **TERMCAP** UNIX environment variable; if it is set to **terminfo**, you must set the **TERMINFO** UNIX environment variable.

NOSORTINDEX

The **NOSORTINDEX** environment variable can be defined to *disable* the default fast-indexing functionality in version 5.0 of IBM Informix OnLine. If the **NOSORTINDEX** environment variable is set, all indexes are built as they were in versions prior to 5.0. (The database server reads through the data pages and adds each index entry as it is encountered in the data row.)

If **NOSORTINDEX** is *not* set, then larger indexes automatically are created differently than in versions prior to 5.0. (For larger tables, those spanning more than 30 pages or containing more than 500 rows, the database server first reads through all of the data pages for a table and extracts the data to be indexed. Before inserting the index entries into the B+ tree, the index entries are sorted, resulting in faster index creation.)

Before you decide whether to set **NOSORTINDEX**, read the discussion of indexing and optimization in the *IBM Informix Guide to SQL: Tutorial*.

Set the **NOSORTINDEX** environment variable as follows:

C shell: `setenv NOSORTINDEX`

Bourne shell: `NOSORTINDEX=
export NOSORTINDEX`

SQLEXEC

The **SQLEXEC** environment variable directs the processes of your application development tool to the appropriate database server. The processes first look for the IBM Informix OnLine database server. Therefore, you must set **SQLEXEC** only if you have both the IBM Informix SE and IBM Informix OnLine database servers installed on your system and you want to access IBM Informix SE.

SQLEXEC must contain the full pathname of the database server, which is found in the **lib** subdirectory of your **\$INFORMIXDIR** directory.

To specify the IBM Informix SE database server, set the **SQLEXEC** environment variable as follows:

C shell: `setenv SQLEXEC $INFORMIXDIR/lib/sqlxec`

Bourne shell: `SQLEXEC=$INFORMIXDIR/lib/sqlxec
export SQLEXEC`

Reset **SQLEXEC** to the IBM Informix OnLine database server as follows:

C shell: `setenv SQLEXEC $INFORMIXDIR/lib/sqlturbo`

Bourne shell: `SQLEXEC=$INFORMIXDIR/lib/sqlturbo
export SQLEXEC`

SQLRM

In an IBM Informix NET or IBM Informix STAR environment, you can configure clients to use a Relay Module. To do this, set the **SQLRM** environment variable to indicate that your IBM Informix product should use a Relay Module instead of a database server when you specify a database on another network server. The Relay Module is similar to the **sqlexec** process on a network server.

If **SQLRM** is set, a Relay Module is used (instead of a database server) to access a database on another server. If **SQLRM** is not set, a Relay Module is not used.

If **SQLRM** is set, the database server looks for the Relay Module directory according to the path specified by **SQLRMDIR**. Set **SQLRM** as follows:

C shell: `setenv SQLRM /usr/rm/sqlrmiittt`

Bourne shell: `SQLRM=/usr/rm/sqlrmiittt`
 `export SQLRM`

where *iii* is a code that indicates the network interface and *ttt* is a code that determines the network transport. For example:

iii	Interface	ttt	Transport
soc	Berkeley Sockets	tcp	TCP/IP
tli	tli	grp	AT&T StarGROUP
usr	user-written	usr	Other user-supplied network

There is a separate Relay Module for each network transport/interface supported in Version 5.0. Each port has a default Relay Module that is displayed when the installation script is run. Contact your Database Administrator for the names of the Relay Modules for your port. For more information on the Relay Module and network protocols, see *IBM Informix NET and IBM Informix STAR Installation and Configuration Guide*.

SQLRMDIR

In an IBM Informix NET or IBM Informix STAR environment, you can set the **SQLRMDIR** environment variable to point to the directory in which all the Relay Module executable files defined by the **SQLRM** environment variable reside. If you do not set **SQLRMDIR**, the default is **\$INFORMIXDIR/lib**.

To specify the path to the Relay Module executable files in **/usr/rm** set **SQLRMDIR** as follows:

C shell: `setenv SQLRMDIR /usr/rm`

Bourne shell: `SQLRMDIR=/usr/rm`
`export SQLRMDIR`

For more information on the Relay Module and network protocols, see *IBM Informix NET and IBM Informix STAR Installation and Configuration Guide*.

TBCONFIG

The **TBCONFIG** environment variable contains the name of the **tbconfig** file that holds the configuration parameters for IBM Informix OnLine. You need to set **TBCONFIG** only if there is more than one IBM Informix OnLine system initialized in your **\$INFORMIXDIR** directory. If you do not set **TBCONFIG**, the default is **tbconfig**.

Each IBM Informix OnLine system has its own **tbconfig** file that must be stored in the **\$INFORMIXDIR/etc** directory. You might prefer to name **tbconfig** so it easily can be related to a specific IBM Informix OnLine system. For example, when the desired filename is **tbconfig3**, set the **TBCONFIG** environment variable as follows:

C shell: `setenv TBCONFIG tbconfig3`

Bourne shell: `TBCONFIG=tbconfig3`
`export TBCONFIG`

UNIX Environment Variables

IBM Informix products also rely on the correct setting of standard UNIX system environment variables. All require that the **PATH** and **TERM** environment variables be set, and some also require that the **TERMCAP** or **TERMINFO** environment variables be set. As with Informix environment variables, you can set UNIX environment variables at the system prompt or in your **.login** or **.cshrc** (C shell) or your **.profile** (Bourne shell) file.

PATH

The **PATH** environment variable tells the shell the order in which to search directories for executable programs. You must include the directory that contains your IBM Informix product in your **PATH** environment variable before you can use the product.

You can specify the correct search path in various ways. Be sure to include a colon between the directory names.

The following example uses the explicit path **/usr/informix**. This path must correspond to the **INFORMIXDIR** setting.

C shell: `setenv PATH $PATH:/usr/informix/bin`

Bourne shell: `PATH=$PATH:/usr/informix/bin`
 `export PATH`

The next example specifies **\$INFORMIXDIR** instead of **/usr/informix**. It tells the shell to search the directories that were specified when **INFORMIXDIR** was set. You might prefer to use this version to ensure that your **PATH** entry does not contradict the path that was set in **INFORMIXDIR**, and so that you do not have to reset **PATH** whenever you change **INFORMIXDIR**.

C shell: `setenv PATH $PATH:$INFORMIXDIR/bin`

Bourne shell: `PATH=$PATH:$INFORMIXDIR/bin`
 `export PATH`

If you set the **PATH** environment variable at the command line instead of in your **.login** or **.cshrc** file, you must include curly braces with the existing **INFORMIXDIR** and **PATH**, as follows:

C shell: `setenv PATH ${INFORMIXDIR}/bin:${PATH}`

TERM

The **TERM** environment variable is used for terminal handling. It enables DB-Access, IBM Informix 4GL, and IBM Informix SQL to recognize and communicate with the terminal you are using. The terminal type specified in the **TERM** setting must correspond to an entry in the **termcap** file or **terminfo** directory. Before you can set the **TERM** environment variable, you must obtain the code for your terminal from the OnLine or SE administrator.

For example, to specify the `vt100` terminal, set the **TERM** environment variable as follows:

C shell: `setenv TERM vt100`

Bourne shell: `TERM=vt100`
 `export TERM`

TERMCAP

The **TERMCAP** environment variable is used for terminal handling. It tells DB-Access, IBM Informix 4GL, and IBM Informix SQL to communicate with the **termcap** file instead of the **terminfo** directory. The **termcap** file contains a list of various types of terminals and their characteristics. If you use **TERMCAP**, you also must set **INFORMIXTERM** to **termcap**.

C shell: `setenv TERMCAP /usr/informix/etc/termcap`

Bourne shell: `TERMCAP=/usr/informix/etc/termcap`
 `export TERMCAP`

TERMINFO

The **TERMINFO** environment variable is used for terminal handling. It is supported only on machines that provide full support for the UNIX System V **terminfo** library.

TERMINFO tells DB-Access, IBM Informix 4GL, and IBM Informix SQL to communicate with the **terminfo** directory instead of the **termcap** file. The **terminfo** directory has subdirectories that contain files pertaining to terminals and their characteristics. If you use **TERMINFO**, you also must set **INFORMIXTERM** to **terminfo**.

C shell: `setenv TERMINFO /usr/lib/terminfo`

Bourne shell: `TERMINFO=/usr/lib/terminfo`
 `export TERMINFO`

Error Handling with SQLCA

In This Chapter	5-3
The SQLCA Record in IBM Informix 4GL	5-5
The sqlca Structure in IBM Informix ESQL/C	5-7
The SQLCA Record in IBM Informix ESQL/COBOL	5-10

In This Chapter

When you execute an SQL statement, the database server always returns a result code, along with other information concerning the operation, in a data structure known as the SQL Communication Area (SQLCA). The SQLCA data structure only stores information about the most recently executed SQL statement. Any 4GL or embedded-language (ESQL) program can check the contents of this structure to ensure that SQL statements execute as anticipated.

You can access the SQLCA data structure with any of the following IBM Informix programming-language products:

- 4GL
- ESQL/C
- ESQL/COBOL

[Figure 5-1](#) summarizes the SQLCA fields available with each IBM Informix programming-language product.

Figure 5-1*Summary table of SQLCA data structure information for IBM Informix products*

Programming Language	Provides Result Code	Provides Details of Statement Execution	Reports Special Conditions
4GL	*STATUS or SQLCA.SQLCODE	SQLCA.SQLERRD[1] through SQLCA.SQLERRD[6]	SQLCA.SQLAWARN[1] through SQLCA.SQLAWARN[8]
ESQL/C	sqlca.sqlcode or SQLCODE	sqlca.sqlerrd[0] through sqlca.sqlerrd[5]	sqlca.sqlwarn.sqlwarn0 through sqlca.sqlwarn.sqlwarn7
ESQL/COBOL	SQLCODE OF SQLCA	SQLERRD[1] OF SQLCA through SQLERRD[6] OF SQLCA	SQLWARN0 OF SQLWARN OF SQLCA through SQLWARN7 OF SQLWARN OF SQLCA

* STATUS is equivalent to SQLCA.SQLCODE only after SQL statements. Other IBM Informix 4GL statements also set STATUS

IBM Informix programming-language products return a result code (see the second column in [Figure 5-1](#)) into the SQLCA data structure after executing every SQL statement except the DECLARE statement. If the result code is a negative value, the statement was not executed successfully. Additionally, the SQLCA data structure provides information about any exceptional conditions or problems that were detected during execution of the statement.

This chapter describes the fields of the SQLCA data structure in the different IBM Informix programming-language products.

The SQLCA Record in IBM Informix 4GL

The IBM Informix 4GL SQLCA record is shown here:

```
DEFINE SQLCA RECORD
  SQLCODE INTEGER,
  SQLERRM CHAR(71),
  SQLERRP CHAR(8),
  SQLERRD ARRAY [6] OF INTEGER,
  SQLAWARN CHAR(8)
END RECORD
```

The following list describes each field in this record:

SQLCODE indicates the result of executing an SQL statement.

It is set as follows:

- To zero for a successful execution of most statements
- To NOTFOUND (defined as 100) for a successfully executed query that returns zero rows or for a FETCH that seeks beyond the end of an active set
- To a negative value for an unsuccessful execution

IBM Informix 4GL sets the global variable STATUS equal to SQLCODE after each SQL statement. However, any subsequent 4GL statement can reset STATUS.

SQLERRM contains the error message (maximum of 71 characters).

SQLERRP is reserved for future use.

SQLERRD is an array of the following six variables of type INTEGER:

SQLERRD[1] is not used at this time.

SQLERRD[2] is the SERIAL value returned or an error code.

SQLERRD[3] is the number of rows processed.

SQLERRD[4] is the estimated CPU cost of the query.

SQLERRD[5] is the offset of error into the SQL statement.

SQLERRD[6] is the rowid of the last row processed.

SQLAWARN	is a character string of length eight whose individual characters signal various warning conditions (as opposed to errors) following the execution of an SQL statement. The characters are blank if no problems or exceptional conditions are detected.
SQLAWARN[1]	is set to w if one or more of the other warning characters has been set to w. If SQLAWARN[1] is blank, you do not have to check the remaining warning characters.
SQLAWARN[2]	is set to w if one or more data items were truncated to fit into a CHAR program variable or if a DATABASE statement selected a database with transactions.
SQLAWARN[3]	is set to w if an aggregate function (SUM, AVG, MAX, or MIN) encountered a null value in its evaluation or if a DATABASE statement selected an ANSI-compliant database.
SQLAWARN[4]	is set to w if a DATABASE statement selected an IBM Informix OnLine database or when the number of items in the <i>select-list</i> of a SELECT clause is not the same as the number of program variables in the INTO clause. In the latter case, the number of values returned by 4GL is the smaller of these two numbers.
SQLAWARN[5]	is set to w if float-to-decimal conversion is used.
SQLAWARN[6]	is set to w when your program executes an 4GL extension to ANSI-compliant standard syntax and the DBANSIWARN environment variable is set or the -ansi option is specified.
SQLAWARN[7]	is reserved for future use.
SQLAWARN[8]	is reserved for future use.

The sqlca Structure in IBM Informix ESQL/C

The ESQL/C `sqlca` structure is defined in the `sqlca.h` file and is shown here. The `sqlca.h` header file is included automatically in an ESQL/C program.

```
#ifndef SQLCA_INCL

#define SQLCA_INCL

struct sqlca_s
{
    long sqlcode;
    char sqlerrm[72]; /* error message parameters */
    char sqlerrp[8];
    long sqlerrd[6];
        /* 0 - estimated number of rows returned */
        /* 1 - serial value after insert or ISAM error code */
        /* 2 - number of rows processed */
        /* 3 - estimated cost */
        /* 4 - offset of the error into the SQL statement */
        /* 5 - rowid after insert */
    struct sqlcaw_s
    {
        char sqlwarn0; /* = W if any of sqlwarn[1-7] = W */
        char sqlwarn1; /* = W if any truncation occurred or
            database has transactions */
        char sqlwarn2; /* = W if a null value returned or
            ANSI database */
        char sqlwarn3; /* = W if no. in select list != no. in into
            list or OnLine backend */
        char sqlwarn4; /* = W if no where clause on prepared update,
            delete or incompatible float format */
        char sqlwarn5; /* = W if non-ANSI statement */
        char sqlwarn6; /* reserved */
        char sqlwarn7; /* reserved */
    } sqlwarn;
    };

extern struct sqlca_s sqlca;

extern long SQLCODE;

#define SQLNOTFOUND 100

#endif /* SQLCA_INCL */
```

The following list describes the fields in this structure:

- sqlcode** indicates the result of executing an IBM Informix ESQL/C statement. It is set as follows:
- To zero for a successful execution of most statements
 - To SQLNOTFOUND (defined as 100 in **sqlca.h**) for a successfully executed query that returns zero rows or for a FETCH that seeks beyond the end of an active set (However, in an ANSI-compliant database, if an INSERT INTO/SELECT statement or a DELETE, UPDATE, or SELECT INTO TEMP statement fails to access any rows, the value of SQLCODE OF SQLCA is set to 100 rather than 0.)
 - To a negative value for an unsuccessful execution

SQLCODE is another name for **sqlca.sqlcode** and is available in ESQL/C files and in C modules that include **sqlca.h**.

sqlerrm contains the error message (maximum of 71 characters).

sqlerrp is reserved for future use.

sqlerrd is an array of six long integers.

sqlerrd(0) is the estimated number of rows returned.

sqlerrd(1) is a SERIAL value returned or an error code.

sqlerrd(2) is the number of rows processed.

sqlerrd(3) is a weighted sum of disk accesses and total rows processed.

sqlerrd(4) is the offset of error into the SQL statement.

sqlerrd(5) is the rowid of the last row processed.

sqlwarn	is a structure containing eight characters whose individual characters signal various warning conditions (as opposed to errors) following the execution of an SQL statement. The characters are blank if no problems are detected.
sqlwarn0	is set to <i>w</i> if one or more of the other warning characters has been set to <i>w</i> . If sqlwarn0 is blank, you do not have to check the remaining warning characters.
sqlwarn1	is set to <i>w</i> if one or more data items were truncated to fit into a character host variable or if a DATABASE statement selected a database with transactions. You can discover which item was truncated by examining the associated indicator variables.
sqlwarn2	is set to <i>w</i> if an aggregate function (SUM, AVG, MAX, MIN) encountered a null value in its evaluation or if a DATABASE statement selected an ANSI-compliant database (with transactions).
sqlwarn3	is set to <i>w</i> if a DATABASE statement selected an IBM Informix OnLine database or when the number of items in the <i>select-list</i> of a SELECT clause is not the same as the number of host variables in the INTO clause. The number of items that IBM Informix ESQL/C returns is the smaller of these two numbers.
sqlwarn4	is set to <i>w</i> if float-to-decimal conversion is used. It is also set to <i>w</i> by the DESCRIBE statement when an UPDATE or DELETE statement is prepared without a WHERE clause. Without a WHERE clause, the UPDATE or DELETE statement applies to the entire table. By checking this variable, you can avoid unintended global changes to your table.

- sqlwarn5** is set to w when your program executes an Informix extension to an ANSI-compliant standard syntax and the **DBANSIWARN** environment variable is defined or the **-ansi** option is specified.
- sqlwarn6** is reserved for future use.
- sqlwarn7** is reserved for future use.

The SQLCA Record in IBM Informix ESQL/COBOL

The IBM Informix ESQL/COBOL SQLCA record is included in each program automatically. These records vary depending on the COBOL compiler.

The SQLCA record for Ryan-McFarland compilers is as follows:

```
77  SQLNOTFOUND PIC S9(10) VALUE 100.
01  SQLCA.
    05  SQLCODEPIC S9(5)    COMPUTATIONAL-4.
    05  SQLERRM.
        49  SQLERRML PIC S9(4)    COMPUTATIONAL-4.
        49  SQLERRMC PIC X(70) .
    05  SQLERRP PIC X(8) .
    05  SQLERRD OCCURS 6 TIMES
        PIC S9(5)    COMPUTATIONAL-4.
    05  SQLWARN.
        10  SQLWARN0 PIC X(1) .
        10  SQLWARN1 PIC X(1) .
        10  SQLWARN2 PIC X(1) .
        10  SQLWARN3 PIC X(1) .
        10  SQLWARN4 PIC X(1) .
        10  SQLWARN5 PIC X(1) .
        10  SQLWARN6 PIC X(1) .
        10  SQLWARN7 PIC X(1) .
```


The SQLCA record for Micro Focus compilers is as follows:

```
77 SQLNOTFOUND PIC S9(10) VALUE 100.
01 SQLCA.
05 SQLCODE      PIC S9(9)    COMPUTATIONAL-5.
05 SQLERRM.
   49 SQLERRML  PIC S9(4)    COMPUTATIONAL-5.
   49 SQLERRMC  PIC X(70) .
05 SQLERRP      PIC X(8) .
05 SQLERRD      OCCURS 6 TIMES
                  PIC S9(9)    COMPUTATIONAL-5.
05 SQLWARN.
   10 SQLWARN0  PIC X(1) .
   10 SQLWARN1  PIC X(1) .
   10 SQLWARN2  PIC X(1) .
   10 SQLWARN3  PIC X(1) .
   10 SQLWARN4  PIC X(1) .
   10 SQLWARN5  PIC X(1) .
   10 SQLWARN6  PIC X(1) .
   10 SQLWARN7  PIC X(1) .
```

The following list describes the components of these records:

SQLCODE indicates the result of executing an SQL statement. It is set as follows:

- To zero for a successful execution of most statements
- To SQLNOTFOUND (defined as 100) for a successfully executed query that returns zero rows or for a FETCH that seeks beyond the end of an active set (However, in an ANSI-compliant database, if an INSERT INTO/SELECT statement or a DELETE, UPDATE, or SELECT INTO TEMP statement fails to access any rows, the value of SQLCODE OF SQLCA is set to 100 rather than 0.)
- To a negative value for an unsuccessful execution

SQLERRM contains the error message.

SQLERRML contains the length of the string.

SQLERRMC denotes the character buffer (maximum of 70 characters).

SQLERRP is reserved for future use.

SQLERRD	is an array of six numerics.
SQLERRD[1]	is the estimated number of rows returned.
SQLERRD[2]	is the SERIAL value returned or an error code.
SQLERRD[3]	is the number of rows processed.
SQLERRD[4]	is a weighted sum of disk accesses and total rows processed.
SQLERRD[5]	is the offset of error into the SQL statement.
SQLERRD[6]	is the rowid of the last row processed.
SQLWARN	is a group item containing eight characters whose individual characters signal various warning conditions (as opposed to errors) following the execution of an SQL statement. The characters are blank if no problems are detected.
SQLWARN0	is set to w if one or more of the other warning characters has been set to w. If SQLWARN0 is blank, you do not have to check the remaining warning characters.
SQLWARN1	is set to w if one or more data items was truncated to fit into a character host variable, a conversion error occurred for a numeric variable, or a DATABASE statement selected a database with transactions. You can discover which item was truncated or incorrectly converted by examining the associated indicator variables.
SQLWARN2	is set to w if an aggregate function (SUM, AVG, MAX, MIN) encountered a null in its evaluation or if a DATABASE statement selected an ANSI-compliant database (with transactions).

SQLWARN3	is set to w if a DATABASE statement selected an IBM Informix OnLine database or when the number of items in the <i>select-list</i> of a SELECT clause is not the same as the number of host variables in the INTO clause. The number of values that IBM Informix ESQL/COBOL returns is the smaller of these two numbers.
SQLWARN4	is set to w if float-to-decimal conversion is used. It is also set to w by the DESCRIBE statement when an UPDATE or DELETE statement is prepared without a WHERE clause. Without a WHERE clause, the UPDATE or DELETE statement applies to the entire table. By checking this variable, you can avoid unintended global changes to your table.
SQLWARN5	is set to w when your program executes an Informix extension to an ANSI-compliant standard syntax and the DBANSIWARN environment variable is defined or the -ansi switch is specified.
SQLWARN6	is reserved for future use.
SQLWARN7	is reserved for future use.

Using Descriptors

In This Chapter	6-3
The System Descriptor Area and the sqlda Structure in ESQL/C	6-4
Using a System Descriptor Area	6-5
The System Descriptor Area	6-7
Using Pointers to an sqlda Structure	6-9
The sqlda.h Header File	6-10
The System Descriptor Area in ESQL/COBOL	6-13
Using a System Descriptor Area	6-13
The System Descriptor Area	6-16

In This Chapter

Use the *system descriptor area* or a structure known as the SQL Descriptor Area (**sqlda**) to hold descriptive information about data in dynamic SQL statements. The following IBM Informix programming-language products use the system descriptor area or the **sqlda** structure when handling dynamic SQL statements:

- IBM Informix ESQL/C
- IBM Informix ESQL/COBOL

IBM Informix ESQL/C can use both the **sqlda** structure and the system descriptor area. The declaration of the **sqlda** structure is stored in the **sqlda.h** header file. IBM Informix ESQL/COBOL can use only the system descriptor area.

This chapter describes the components of the system descriptor area and of the **sqlda** structure. It tells how these fields are used in the three ESQL products and briefly discusses the SQL statements that support dynamic memory allocation. See the discussion of dynamic SQL in the *IBM Informix Guide to SQL: Tutorial* and the chapter on dynamic management in your ESQL product manual for more information.

The System Descriptor Area and the sqlda Structure in ESQL/C

In ESQL/C, you can allocate memory dynamically by using either a system descriptor area or an **sqlda** structure.

You use the system descriptor area when you issue the **ALLOCATE DESCRIPTOR**, **GET DESCRIPTOR**, and **SET DESCRIPTOR** statements. These statements let you determine the contents of a prepared statement at run time and allocate memory dynamically. They also allow you to create **WHERE** clauses for statements that receive **WHERE**-clause values at run time.

- The **ALLOCATE DESCRIPTOR** statement allocates memory for a system descriptor area that is identified by a descriptor. It creates a place in memory to hold information obtained by a **DESCRIBE** statement or information about the **WHERE** clause of a statement. (The **DEALLOCATE DESCRIPTOR** statement frees the allocated system descriptor area.)
- The **GET DESCRIPTOR** statement allows you to determine how many values have been described in a system descriptor area, determine the characteristics of each of the columns or expressions described in the system descriptor area, or copy a value out of the system descriptor area and into a host variable after a **FETCH** statement.
- The **SET DESCRIPTOR** statement assigns values to a system descriptor area identified by a descriptor.

The **DESCRIBE** statement returns information about a prepared statement before you execute it. In ESQL/C, the information can be stored in a system descriptor area (identified by a descriptor) or in an **sqlda** structure (identified by a pointer).

Using a System Descriptor Area

The X/Open implementation of dynamic SQL lets you allocate space in memory with a system descriptor area. Thus, you can use a standardized descriptor area structure and write more portable code.

You can allocate a system descriptor area identified by a *descriptor* or *descriptor variable* and specify its size with the `ALLOCATE DESCRIPTOR` statement. You can direct the output of a `DESCRIBE` statement on a `SELECT` or `INSERT` statement to a system descriptor area. You also can set the contents of a system descriptor explicitly. You can retrieve information stored in such system descriptor areas by executing a `GET DESCRIPTOR` statement following a `DESCRIBE` statement.

Use the `SET DESCRIPTOR` statement to assign values to an allocated system descriptor area. Space for the `DATA` field of the system descriptor area is allocated automatically by a `DESCRIBE` or `SET DESCRIPTOR` statement.

You can use the system descriptor area to provide storage for values returned from a `FETCH` statement. Release memory associated with the system descriptor area with the `DEALLOCATE DESCRIPTOR` statement.

Other statements that support the use of a system descriptor area are `EXECUTE`, `OPEN`, and `PUT`. For more information on dynamic SQL and system descriptors, see the discussion of `ALLOCATE DESCRIPTOR` on page 7-13, `DEALLOCATE DESCRIPTOR` on page 7-105, `GET DESCRIPTOR` on page 7-169, `SET DESCRIPTOR` on page 7-293, `DESCRIBE` on page 7-125, and `FETCH` on page 7-153. See also `EXECUTE` on page 7-142, `OPEN` on page 7-207, and `PUT` on page 7-230.

A system descriptor area has a field for the *count* of values returned by a SELECT statement or inserted into an INSERT statement. It also has a set of fields for each value or item that is input or returned. [Figure 6-1](#) illustrates a descriptor area for two values.

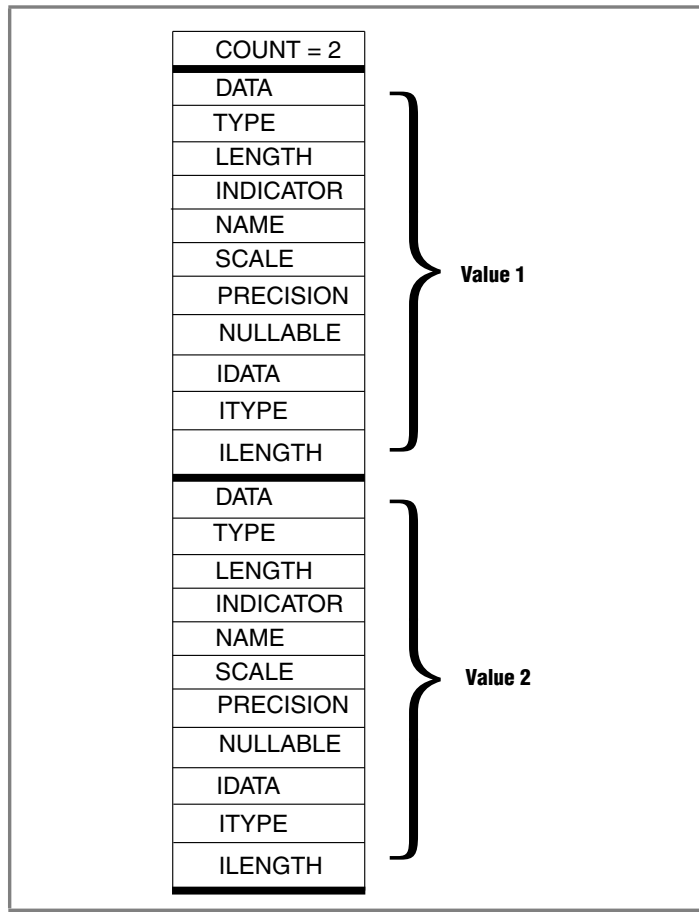


Figure 6-1
A system descriptor area for two values

The System Descriptor Area

These are the fields in the system descriptor area.

Field	Description
COUNT	is the number of VALUES, items, or <i>occurrences</i> in the system descriptor area, as follows: After ALLOCATE DESCRIPTOR, it is set to the number of occurrences; DESCRIBE sets it to the number of values in the SELECT or INSERT list. (This can be obtained using GET DESCRIPTOR.) If you are using a system descriptor area to hold parameters for a PUT, OPEN, or EXECUTE statement, you must set the COUNT field to the number of parameters.
DATA	is the data. It can be a host variable or a numeric literal, character string literal, DATETIME literal, or INTERVAL literal.
TYPE	is a short integer corresponding to the data type being transferred. The integer correspondences are defined in Figure 6-2 and in Figure 6-3 .
LENGTH	is a short integer that gives the size in bytes of CHAR type data, the encoded qualifiers of DATETIME or INTERVAL data, or the size of a DECIMAL or MONEY value.
INDICATOR	is a short integer indicator variable. INDICATOR can contain two values: 0, meaning there is non-null data in the DATA field, and -1, meaning there is NULL data in the DATA field.
NAME	is a character string containing the column name or display label being transferred.
SCALE	is defined only for the DECIMAL or MONEY data type. After a DESCRIBE statement is executed, it contains the scale of the column. In a SET DESCRIPTOR statement, it must be set to indicate the scale of the value in the DATA field.
PRECISION	is defined only for the DECIMAL or MONEY data type. After a DESCRIBE statement is executed, it contains the precision of the column. Otherwise, it must be set to indicate the precision of the value in the DATA field.

(1 of 2)

Field	Description
NULLABLE	<p>specifies whether a resulting column can contain a null value after a DESCRIBE statement is executed, as follows: 1, meaning the column allows null values, and 0, meaning the column does <i>not</i> allow null values.</p> <p>Before an EXECUTE statement or a dynamic OPEN statement is executed, it must be set to 1 to indicate that an indicator value is specified in the INDICATOR field, and to 0 if it is not specified. (When executing a dynamic FETCH statement, the NULLABLE field is ignored.)</p>
IDATA	is user-defined indicator data or the name of a host variable that contains indicator data for the DATA field.
ITYPE	is the data type for a user-defined, short-integer indicator. The integer correspondences are defined in Figure 6-2 and in Figure 6-3 .
ILENGTH	is the length, in bytes, of the user-defined indicator.

(2 of 2)

[Figure 6-2](#) and [Figure 6-3](#) show the values for TYPE and ITYPE in X/Open mode and in standard mode.

Figure 6-2
Values for the TYPE and ITYPE fields for X/Open SQL

Data Type	Integer
CHARACTER	1
DECIMAL	3
INTEGER	4
SMALLINT	5
FLOAT	6

Figure 6-3*Values for the TYPE and ITYPE fields when not using X/Open SQL*

Data Type	Integer
CHARACTER	0
DECIMAL	5
INTEGER	2
SMALLINT	1
FLOAT	3
SMALLFLOAT	4
SERIAL	6
DATE	7
MONEY	8
DATETIME	10
BYTE	11
TEXT	12
VARCHAR	13
INTERVAL	14
FILE	116

Using Pointers to an `sqllda` Structure

You use a pointer to the `sqllda` structure when your ESQL/C program performs the dynamic memory allocation and you allocate the memory for each of the dynamic variables in your code.

The DESCRIBE statement sets a pointer to an `sqllda` structure and describes the data that is retrieved when the described statement identifier is executed.

You can use this information in statements that support pointers to the `sqllda` structure, such as EXECUTE, FETCH, OPEN, and PUT.

The allocation of memory for an `sqllda` structure is done at run time. When all of its components are fully defined, the `sqllda` structure points to the beginning of a sequence of `sqlvar_struct` structures that contain the necessary information for each variable in the set.

For more information on dynamic SQL and the use of pointers, see the discussions of `DESCRIBE` on page 7-125, `EXECUTE` on page 7-142, `FETCH` on page 7-153, `OPEN` on page 7-207, and `PUT` on page 7-230.

The `sqllda.h` Header File

The declaration of the `sqllda` structure is stored in the `sqllda.h` header file, as shown in Figure 6-4. It comprises an `sqllda` structure that points to an `sqlvar_struct` structure.

Figure 6-4
The `sqllda.h` header file

```
struct sqlvar_struct
{
short sqltype;
short sqlllen;
char *sqldata;
short *sqlind;
char *sqlname;
char *sqlformat;
short sqlitype;
short sqlilen;
char *sqlidata;
}

struct sqllda
{
short sqld;
struct sqlvar_struct *sqlvar;
char *desc_name;
short desc_occ;
struct sqllda *desc_next;
}
```



Tip: Because of the additional fields in the `sqllda` structure in Version 5.0, you must recompile old applications that use this structure to make use of the new libraries.

The sqlda Structure

The **sqlda** structure specifies the number of **sqlvar** occurrences that contain descriptions.

- After a DESCRIBE statement is executed, **sqld** is set to the number of the select-list columns and expressions or insert-list columns, if the described statement is a cursor specification; or to zero, if the described statement is not a cursor specification.
- Before an EXECUTE statement or a dynamic OPEN statement is executed, **sqld** must be set to the number of input values.
- Before a dynamic FETCH statement is executed, **sqld** must correspond to the number of output values.

Figure 6-5 shows the fields in the **sqlda** structure.

Figure 6-5
Fields in the sqlda structure

Field	Description
sqld	is a short integer representing the number of values in the sqlvar array.
sqlvar	is a pointer to an array of sqlvar_struct structures.

The sqlvar_struct Structure

The **sqlvar_struct** structure holds a group of fields that contains one of the following pieces of information, depending on the statement:

- A description of a resulting column of the cursor specification
- An input value and its description
- An output value and its description

Figure 6-6 shows the fields in the **sqlvar_struct** structure.

Figure 6-6
Fields in the `sqlvar_struct` structure

Field	Description
sqltype	is a short integer corresponding to the data type being transferred.
sqllen	is a short integer that gives the size, in bytes, of CHAR type data or the qualifier of a DATETIME or INTERVAL value.
sqldata	is a pointer to the character data.
sqlind	is a pointer to a short integer indicator variable.
sqlname	is a pointer to a character array containing the column name or display label being transferred.
sqlformat	is a character field reserved for future use.
sqlitype	is a short integer indicator variable type. The integer correspondences are defined in the sqltypes.h header file.
sqlilen	is a short integer indicator length in bytes.
sqlidata	is a pointer to character indicator data.

The System Descriptor Area in ESQL/COBOL

In ESQL/COBOL, you can allocate memory dynamically by using a system descriptor area.

You use the system descriptor area when you use the `ALLOCATE DESCRIPTOR`, `GET DESCRIPTOR`, and `SET DESCRIPTOR` statements. These statements let you determine the contents of a prepared statement at run time and allocate memory dynamically. They also let you create `WHERE` clauses for statements that receive `WHERE`-clause values at run time.

- The `ALLOCATE DESCRIPTOR` statement allocates memory for a system descriptor area that is identified by a descriptor. It creates a place in memory to hold information obtained by a `DESCRIBE` statement or information about the `WHERE` clause of a statement. (The `DEALLOCATE DESCRIPTOR` statement frees the allocated system descriptor area.)
- The `GET DESCRIPTOR` statement allows you to determine how many values have been described in a system descriptor area, determine the characteristics of each of the columns or expressions described in the system descriptor area, or copy a value out of the system descriptor area and into a host variable after a `FETCH` statement.
- The `SET DESCRIPTOR` statement assigns values to a system descriptor area identified by a descriptor.

The `DESCRIBE` statement returns information about a prepared statement before you execute it. The information is stored in a system descriptor area.

Using a System Descriptor Area

The X/Open implementation of dynamic SQL lets you allocate space in memory with a system descriptor area. Thus, you can use a standardized system descriptor area and write more portable code.

You can allocate a system descriptor area identified by a *descriptor* or *descriptor variable* and specify its size with the `ALLOCATE DESCRIPTOR` statement. Only system descriptor areas that have been allocated with the `ALLOCATE DESCRIPTOR` statement can be used in a `DESCRIBE` statement in ESQL/COBOL.

You can direct the output of a DESCRIBE statement on a SELECT or INSERT statement to a system descriptor area. You also can set the contents of a system descriptor explicitly. You can retrieve information stored in such system descriptor areas by executing a GET DESCRIPTOR statement following a DESCRIBE statement.

Use the SET DESCRIPTOR statement to assign values to an allocated system descriptor area. Space for the DATA field of the system descriptor area is allocated automatically by a DESCRIBE or SET DESCRIPTOR statement.

You can use the system descriptor area to provide a storage area for values returned from a FETCH statement. Release memory associated with the system descriptor area with the DEALLOCATE DESCRIPTOR statement.

Other statements that support the use of a system descriptor area are EXECUTE, OPEN, and PUT. For more information on dynamic SQL and system descriptors, see the discussion of ALLOCATE DESCRIPTOR on page 7-13, DEALLOCATE DESCRIPTOR on page 7-105, GET DESCRIPTOR on page 7-169, SET DESCRIPTOR on page 7-293, DESCRIBE on page 7-125, and FETCH on page 7-153. See also EXECUTE on page 7-142, OPEN on page 7-207, and PUT on page 7-230.

A system descriptor area has a field for the *count* of values returned by a SELECT statement or inserted into an INSERT statement. It also has a set of fields for each value or item that is input or returned. Figure 6-7 illustrates a descriptor area for two values.

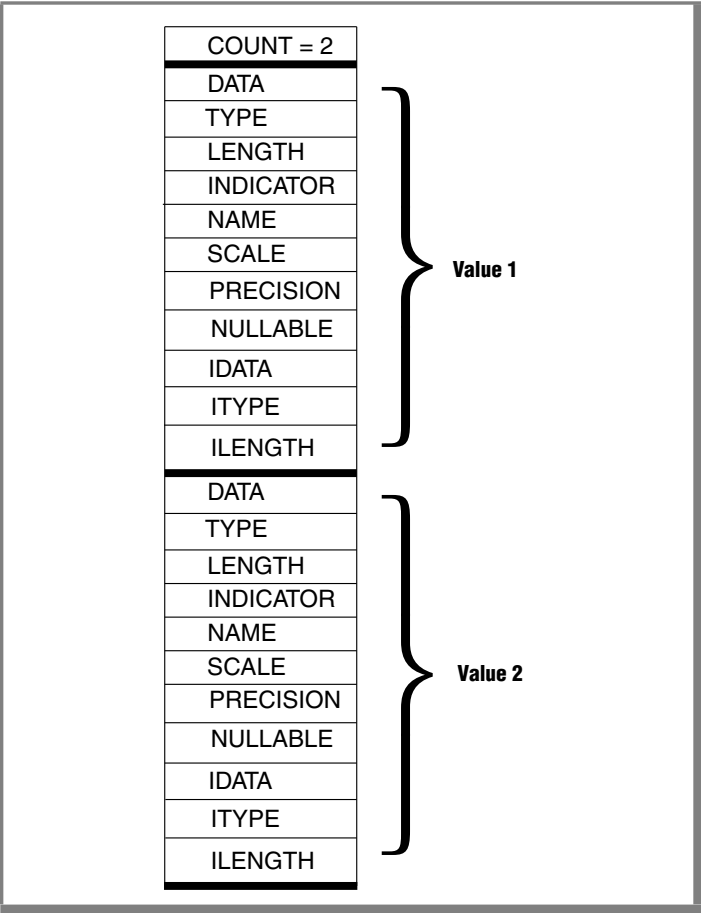


Figure 6-7
A system descriptor area for two values

The System Descriptor Area

These are the fields in the system descriptor area.

Field	Description
COUNT	is the number of VALUES, items, or <i>occurrences</i> in the system descriptor area, as follows: After ALLOCATE DESCRIPTOR, it is set to the number of occurrences; DESCRIBE sets it to the number of values in the SELECT or INSERT list. (This can be obtained using GET DESCRIPTOR.) If you are using a system descriptor area to hold parameters for a PUT, OPEN, or EXECUTE statement, you must set the COUNT field to the number of parameters.
DATA	is the data. It can be a host variable or a numeric literal, character string literal, DATETIME literal, or INTERVAL literal.
TYPE	is a short integer corresponding to the data type being transferred. The integer correspondences are defined in Figure 6-8 and in Figure 6-9 .
LENGTH	is a short integer that gives the size in bytes of CHAR type data, the encoded qualifiers of DATETIME or INTERVAL data, or the size of a DECIMAL or MONEY value.
INDICATOR	is a short integer indicator variable. INDICATOR can contain two values: 0, meaning there is non-null data in the DATA field, and -1, meaning there is NULL data in the DATA field.
NAME	is a character string containing the column name or display label being transferred.
SCALE	is defined only for the DECIMAL or MONEY data type. After a DESCRIBE statement is executed, it contains the scale of the column. In a SET DESCRIPTOR statement, it must be set to indicate the scale of the value in the DATA field.
PRECISION	is defined only for the DECIMAL or MONEY data type. After a DESCRIBE statement is executed, it contains the precision of the column. Otherwise, it must be set to indicate the precision of the value in the DATA field.

(1 of 2)

Field	Description
NULLABLE	<p>specifies whether a resulting column can contain a null value after a DESCRIBE statement is executed, as follows: 1, meaning the column allows null values, and 0, meaning the column does <i>not</i> allow null values.</p> <p>Before an EXECUTE statement or a dynamic OPEN statement is executed, it must be set to 1 to indicate that an indicator value is specified in the INDICATOR field, and to 0 if it is not specified. (When executing a dynamic FETCH statement, the NULLABLE field is ignored.)</p>
IDATA	is user-defined indicator data or the name of a host variable that contains indicator data for the DATA field.
ITYPE	is the data type for a user-defined, short-integer indicator. The integer correspondences are defined in Figure 6-8 and Figure 6-9 .
ILENGTH	is the length, in bytes, of the user-defined indicator.

(2 of 2)

[Figure 6-8](#) and [Figure 6-9](#) show the values for TYPE and ITYPE in X/Open mode and in standard mode.

Figure 6-8
Values for the TYPE and ITYPE fields for X/Open SQL

Data Type	Integer
CHARACTER	1
DECIMAL	3
INTEGER	4
SMALLINT	5
FLOAT	6

Figure 6-9

Values for the TYPE and ITYPE fields when not using X/Open SQL

Data Type	Integer
CHARACTER	0
DECIMAL	5
INTEGER	2
SMALLINT	1
FLOAT	3
SMALLFLOAT	4
SERIAL	6
DATE	7
MONEY	8
DATETIME	10
BYTE	11
TEXT	12
VARCHAR	13
INTERVAL	14
FILE	116

Syntax

In This Chapter	7-9
SQL Statements	7-9
Data Definition Statements	7-10
Data Manipulation Statements	7-10
Cursor Manipulation Statements	7-10
Dynamic Management Statements	7-11
Data Access Statements	7-11
Data Integrity Statements	7-11
Query Optimization Information Statements	7-11
Stored Procedure Statements	7-12
Auxiliary Statements	7-12
ALLOCATE DESCRIPTOR	7-13
The WITH MAX Clause	7-15
ALTER INDEX	7-17
The TO CLUSTER Option	7-18
The TO NOT CLUSTER Option	7-19
ALTER TABLE	7-20
DEFAULT Clause	7-23
Subset of Constraint-Definition Option	7-25
BEGIN WORK	7-37
CHECK TABLE	7-39
CLOSE	7-41
CLOSE DATABASE	7-44
COMMIT WORK	7-46
CREATE AUDIT	7-47
CREATE DATABASE	7-49
Designating Buffered Logging	7-52
Designating an ANSI-Compliant Database	7-52
Designating an ANSI-Compliant IBM Informix SE Database	7-53

CREATE INDEX	7-54
UNIQUE Option	7-55
CLUSTER Option	7-55
Composite Indexes	7-56
The ASC and DESC Keywords	7-56
CREATE PROCEDURE	7-58
DBA Option	7-59
Subset of SQL Data Types Allowed in the Parameter List	7-60
Subset of SQL Statements Allowed in the Statement Block.	7-64
CREATE PROCEDURE FROM	7-67
CREATE SCHEMA	7-68
CREATE SYNONYM	7-70
Synonyms with the Same Name.	7-72
CREATE TABLE	7-75
Limits on Constraint Definitions	7-77
Adding or Dropping Constraints	7-77
Enforcing Primary Key, Unique, and Referential Constraints	7-78
Constraint Names	7-78
The DEFAULT Clause	7-81
Specifying NOT NULL in a Column Definition	7-83
Defining a Column as Unique	7-85
The CHECK Clause	7-89
Subset of Column-Definition Option	7-91
Subset of Constraint-Definition Option	7-91
WITH NO LOG Option for Temporary Tables	7-91
The IN dbspace Clause	7-92
Extent Option	7-94
LOCK MODE Clause	7-95
The IN pathname Option	7-96
CREATE VIEW	7-97
DATABASE	7-101
DEALLOCATE DESCRIPTOR.	7-105
DECLARE	7-107
Select Cursor	7-110
Update Cursor.	7-111
Insert Cursor	7-111
Sequential Cursor.	7-111
Scroll Cursor	7-112
Hold Cursor	7-112
Subset of the SELECT Statement Associated with an Update Cursor	7-115

Locking with an Update Cursor7-115
Using FOR UPDATE with a List of Columns7-116
Using an Insert Cursor with Hold.7-120
DELETE7-122
CURRENT OF Clause7-123
DESCRIBE7-125
DROP AUDIT7-131
DROP DATABASE7-132
DROP INDEX7-134
DROP PROCEDURE7-136
DROP SYNONYM7-137
DROP TABLE7-139
DROP VIEW7-141
EXECUTE7-142
USING Clause7-144
EXECUTE IMMEDIATE7-147
Restricted Statement Types7-148
EXECUTE PROCEDURE7-150
FETCH7-153
Row Numbers7-156
How the Database Server Stores Rows7-156
Using the INTO Clause of SELECT7-157
Using the INTO Clause of FETCH7-158
Using a System Descriptor7-158
FLUSH7-162
Counting Total and Pending Rows7-164
FREE7-165
GET DESCRIPTOR7-169
Using the COUNT Keyword7-171
VALUE Clause7-172
GRANT7-175
INFO7-185
Displaying Tables, Columns, and Indexes7-186
Displaying Privileges, References, and Status.7-187
INSERT7-189
Value and Column Type Compatibility7-195
Inserting Values into SERIAL Columns7-196
Using Functions in the VALUES Clause.7-196
Inserting Nulls with the VALUES Clause7-196

LOAD	7-199
The LOAD FROM File	7-200
DELIMITER Clause	7-203
INSERT INTO Clause	7-203
LOCK TABLE	7-204
OPEN	7-207
Naming Variables in USING	7-211
USING SQL DESCRIPTOR Clause	7-212
OUTPUT	7-216
PREPARE	7-219
Statement Identifier	7-220
Releasing a Statement Identifier	7-221
Statement Text	7-222
Permitted Statements	7-223
PUT	7-231
Using Constant Values in INSERT	7-233
Naming Program Variables in INSERT	7-234
Naming Program Variables in PUT	7-235
Using a System Descriptor Area	7-235
Using an sqlda Structure	7-236
Counting Total and Pending Rows	7-238
RECOVER TABLE	7-239
RENAME COLUMN	7-242
RENAME TABLE	7-244
REPAIR TABLE	7-246
REVOKE	7-248
ROLLBACK WORK	7-255
ROLLFORWARD DATABASE	7-257
SELECT	7-259
Allowing Duplicates	7-262
Expressions in the Select List	7-262
Using a Display Label	7-265
INTO Clause with Indicator Variables	7-266
INTO Clause with Cursors	7-267
Preparing a SELECT...INTO Query	7-268
Using Array Variables with the INTO Clause	7-268
Error Checking	7-269
AS Keyword with Table Aliases	7-272
Using a Condition in the WHERE Clause	7-272
Using a Join in the WHERE Clause	7-278

Using Select Numbers.7-281
Nulls in the GROUP BY Clause7-282
Ordering by a Derived Column7-285
Ascending and Descending Orders7-285
Nulls in the ORDER BY Clause7-285
Nested Ordering7-285
Using Select Numbers.7-286
ORDER BY Clause with DECLARE7-286
INTO TEMP Clause and INTO.7-287
WITH NO LOG Option7-288
Restrictions on a Combined SELECT.7-288
Duplicate Rows in a Combined SELECT7-289
SET CONSTRAINTS7-290
SET DEBUG FILE TO7-292
SET DESCRIPTOR7-294
COUNT Option7-296
VALUE Option7-297
SET EXPLAIN7-302
SET ISOLATION7-308
SET LOCK MODE7-312
SET LOG7-314
SET OPTIMIZATION7-316
START DATABASE7-318
UNLOAD7-320
UNLOAD TO File7-321
DELIMITER Clause7-323
UNLOCK TABLE7-324
UPDATE7-326
Selecting All Columns with the Set Clause.7-330
Subset of Expressions Allowed in the SET Clause7-330
Subset of SELECT Statements Allowed in the SET Clause7-330
Single Columns Paired to Single Expressions7-330
Multiple Columns Equal to Multiple Expressions7-331
UPDATE STATISTICS7-336
WHENEVER7-338

Segments	7-345
Condition	7-346
Relational-Operator Condition	7-349
BETWEEN Condition	7-350
IN Condition	7-351
IS NULL Condition	7-352
LIKE and MATCHES Condition	7-353
Subset of a SELECT Allowed in a Subquery	7-356
IN Subquery	7-356
EXISTS Subquery	7-357
ALL/ANY/SOME Subquery	7-358
Constraint Name	7-361
Database Name	7-363
Data Type	7-366
DATETIME Field Qualifier	7-369
Expression	7-371
Using Subscripts on Character Columns	7-375
Using Rowids	7-375
Using the At Sign	7-376
Quoted String as Expression	7-378
USER Function.	7-378
SITENAME and DBSERVERNAME Functions	7-379
Literal Number as Expression	7-380
TODAY Function	7-380
CURRENT Function	7-380
Literal DATETIME as an Expression	7-382
Literal INTERVAL as an Expression	7-382
UNITS Keyword	7-383
DAY, MONTH, WEEKDAY, and YEAR Functions.	7-386
DATE Function	7-388
EXTEND Function	7-388
MDY Function	7-389
LENGTH Function	7-390
HEX Function	7-391
ROUND Function.	7-392
TRUNC Function	7-393
Subset of Expressions Allowed in an Aggregate Expression	7-395
Including or Excluding Duplicates in the Row Set	7-395
COUNT(*) Keyword.	7-395
AVG Keyword	7-396
MAX Keyword.	7-396

MIN Keyword7-396
SUM Keyword7-397
COUNT Keyword7-397
Summary of Aggregate Function Behavior7-397
Error Checking with Aggregate Functions7-398
Identifier.7-400
Using Keywords as Column Names7-404
Using ALL, DISTINCT, or UNIQUE as a Column Name7-404
Using INTERVAL or DATETIME as a Column Name7-405
Using rowid as a Column Name7-406
Using AS with Column Labels7-407
Using AS with Table Aliases7-408
Using CURRENT, DATETIME, INTERVAL, and NULL in INSERT.7-410
Using NULL and SELECT in a Condition7-410
Using ON, OFF, or PROCEDURE with TRACE7-411
Using GLOBAL as a Variable Name7-411
Index Name7-414
INTERVAL Field Qualifier7-415
Literal DATETIME7-417
Literal INTERVAL7-420
Literal Number7-423
Procedure Name7-425
Quoted String7-427
Relational Operator7-430
Synonym Name7-433
Table Name7-435
View Name7-439

In This Chapter

This chapter covers the purpose, syntax and usage of SQL statements and in addition, discusses elements of the segments that are common to more than one SQL statement.

SQL Statements

SQL statements are divided into the following categories:

- Data definition statements
- Data manipulation statements
- Cursor manipulation statements
- Dynamic management statements
- Data access statements
- Data integrity statements
- Query optimization information statements
- Stored procedure statements
- Auxiliary statements

The specific statements contained in each category are listed below.

Data Definition Statements

ALTER INDEX	CREATE VIEW
ALTER TABLE	DATABASE
CLOSE DATABASE	DROP DATABASE
CREATE DATABASE	DROP INDEX
CREATE INDEX	DROP PROCEDURE
CREATE PROCEDURE	DROP SYNONYM
CREATE PROCEDURE FROM	DROP TABLE
CREATE SCHEMA	DROP VIEW
CREATE SYNONYM	RENAME COLUMN
CREATE TABLE	RENAME TABLE

Data Manipulation Statements

INSERT	SELECT
DELETE	UNLOAD
LOAD	UPDATE

Cursor Manipulation Statements

CLOSE	FLUSH
DECLARE	OPEN
FETCH	PUT

Dynamic Management Statements

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

Data Access Statements

GRANT	SET ISOLATION
LOCK TABLE	SET LOCK MODE
REVOKE	UNLOCK TABLE

Data Integrity Statements

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

Query Optimization Information Statements

SET EXPLAIN	UPDATE STATISTICS
SET OPTIMIZATION	

Stored Procedure Statements

EXECUTE PROCEDURE

SET DEBUG FILE TO

Auxiliary Statements

INFO

WHENEVER

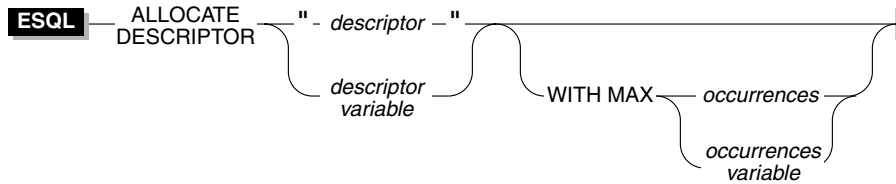
OUTPUT

ALLOCATE DESCRIPTOR

Purpose

Use the ALLOCATE DESCRIPTOR statement to allocate memory for a system descriptor area that is identified by a *descriptor* or *descriptor variable*. Use it to create a place in memory to hold information obtained by a DESCRIBE statement or to hold information about the WHERE clause of a statement.

Syntax



descriptor is a quoted string that identifies the system descriptor area. The *descriptor* must conform to the same rules as any identifier, as described in the Identifier segment on page 7-399.

descriptor variable is an embedded variable name that identifies the system descriptor area being allocated. The *descriptor variable* must conform to the same rules as any identifier, as described in the Identifier segment on page 7-399.

occurrences is an unsigned INTEGER that specifies a value greater than 0. It is the number of items that can be held by the system descriptor area.

occurrences variable is an embedded variable name that specifies a value greater than 0. Its data type must be INTEGER or SMALLINT. It is the number of items that can be held by the system descriptor area.

Usage

The ALLOCATE DESCRIPTOR statement creates a system descriptor area identified by *descriptor* or *descriptor variable*.

A system descriptor area contains one or more *item descriptors*. Each item descriptor holds a data value that can be sent to or received from the database server. The item descriptors also contain information about the database such as type, length, scale, precision, nullability, and so on.

The *occurrences* or *occurrences variable* specifies the number of item descriptors desired in the system *descriptor* or *descriptor variable*.

Initially, all fields of the item descriptor area are undefined. The COUNT is set to the number of occurrences specified. The TYPE, LENGTH, and other information in the item descriptor are set when a DESCRIBE statement is executed using the system descriptor. The DESCRIBE statement also allocates memory for the DATA field in each item descriptor, based on the TYPE and LENGTH information. See [Chapter 6, "Using Descriptors,"](#) for more information.

If a *descriptor* or *descriptor variable* with the same name is already allocated, the system returns an error.

The WITH MAX Clause

You can use the optional WITH MAX *occurrences* clause to indicate the number of value descriptors you need. This number must be greater than zero. If the WITH MAX clause is not specified, a default value of 100 is used for *occurrences*.

The following examples show the ALLOCATE DESCRIPTOR statement for three programming languages. All show the WITH MAX *occurrences* clause.

In each pair, the first example uses an embedded variable name and the second example uses a quoted string to identify the system descriptor area to be allocated. The WITH MAX *occurrences* clause alternately uses an embedded variable name and the unsigned INTEGER 3.

Figure 7-1

Sample ALLOCATE DESCRIPTOR statements in IBM Informix ESQL/C

```
$allocate descriptor $descname with max $occ;
$allocate descriptor "desc1" with max 3;
```

Figure 7-2

Sample ALLOCATE DESCRIPTOR statements in IBM Informix ESQL/COBOL

```
EXEC SQL ALLOCATE DESCRIPTOR :DESCNAME WITH MAX :OCC END-EXEC
EXEC SQL ALLOCATE DESCRIPTOR "DESC1" WITH MAX 3 END-EXEC
```

References

In this manual, see the following statements: DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR, and [Chapter 6, “Using Descriptors.”](#)

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of dynamic SQL.

ALTER INDEX

Purpose

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

Syntax

```

+ ALTER INDEX Index Name  
p. 7-413 TO _____ CLUSTER _____
                                     NOT
  
```

Usage

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

SE

You cannot use a ROLLBACK WORK statement to undo an ALTER INDEX statement. If you roll back a transaction that contains an ALTER INDEX statement, the index remains altered; you do not receive an error message.

If you are using IBM Informix SE and have an audit trail on the table, you cannot use the ALTER INDEX statement. If you want to change an index on an audited table, you must first drop the audit on the table, alter the index, and create a new audit for the table. ♦

The TO CLUSTER Option

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

The following example shows how the ALTER INDEX TO CLUSTER statement is used to physically order the rows in the **orders** table. 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 the entire file to be rewritten. This process can take a long time and it requires sufficient disk space to maintain two copies of the table.

While a table is being clustered, the table is locked in EXCLUSIVE MODE. If another process is using the table to which *index name* belongs, the database server cannot execute the ALTER INDEX statement with the TO CLUSTER option; it returns an error unless lock mode is set to WAIT. (If 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. You can recluster the table by issuing another ALTER INDEX TO CLUSTER statement on the clustered index. You do not need to drop a clustered index before issuing another ALTER INDEX TO CLUSTER statement on a currently clustered index.

The TO NOT CLUSTER Option

The NOT option drops the cluster attribute on *index name* without affecting the physical table. Since there can be only one clustered index per table, you must use the NOT option to release the cluster attribute from one index before you assign it to another. For example, the following series of statements illustrates how clustering is removed from one index and the table is physically reclustered by a second index.

```
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

In this chapter, see the following statements: CREATE INDEX and CREATE TABLE.

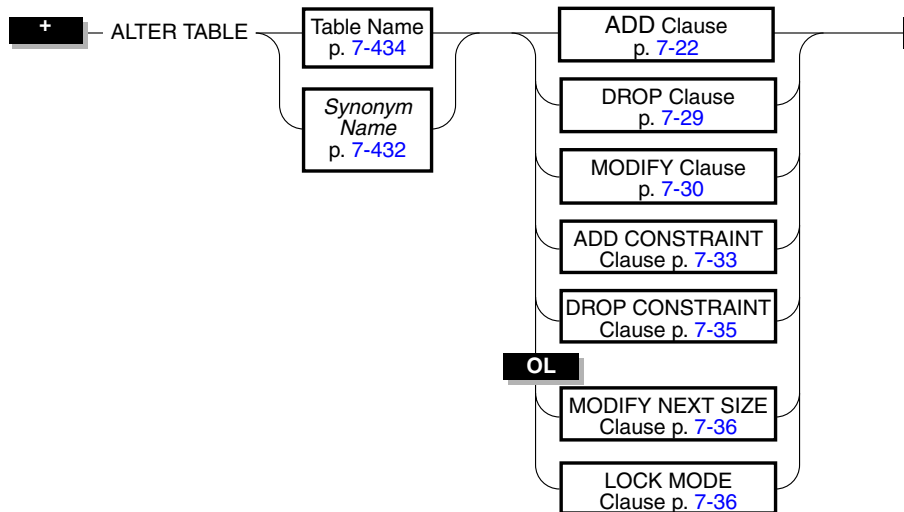
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of clustered indexes.

ALTER TABLE

Purpose

Use the ALTER TABLE statement to add a column to or delete a column from a table, modify the data constraints placed on a column, add a constraint to a column or a composite list of columns, and drop a constraint associated with a column or a composite list of columns.

Syntax



Usage

You must own *table name*, have DBA status, or be granted the Alter privilege on the specified table to use the ALTER TABLE statement. You cannot alter a temporary table.

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

DB

ESQL

DB**ESQL**

To drop a constraint, you must have DBA status, own the table, or be granted Alter privilege on that table.

To drop a constraint in a database, you must have 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. ♦

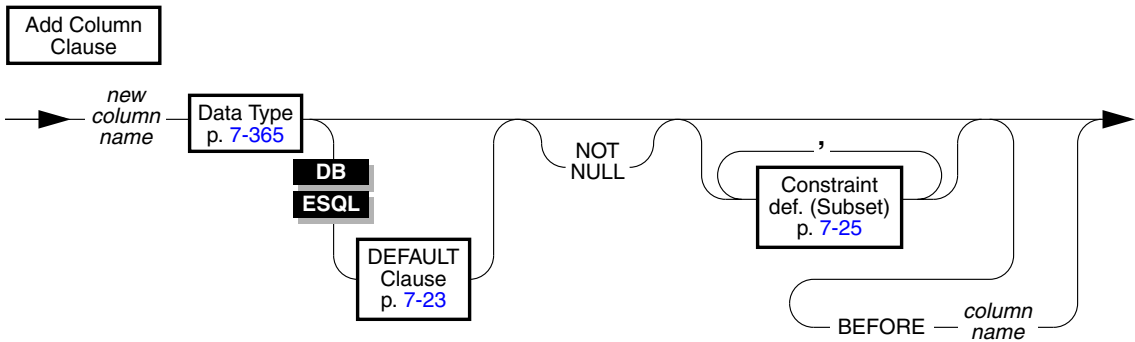
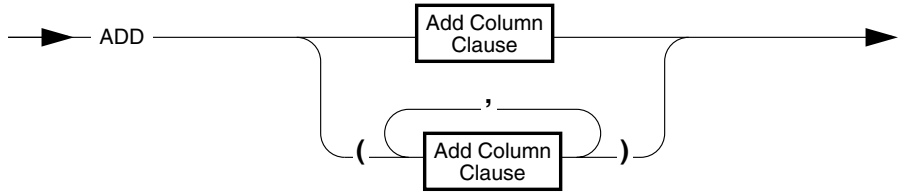
Altering a table on which a view depends may invalidate the view.

You can use one or more of the ADD, DROP, MODIFY, ADD CONSTRAINT, or DROP CONSTRAINT clauses, and you can place them in any order. The actions are performed in the order specified. If any of the actions fail, the entire operation is cancelled.

SE

You cannot use a ROLLBACK WORK statement to undo an ALTER TABLE statement. If you roll back a transaction that contains an ALTER TABLE statement, the table remains altered; you do not receive an error message. ♦

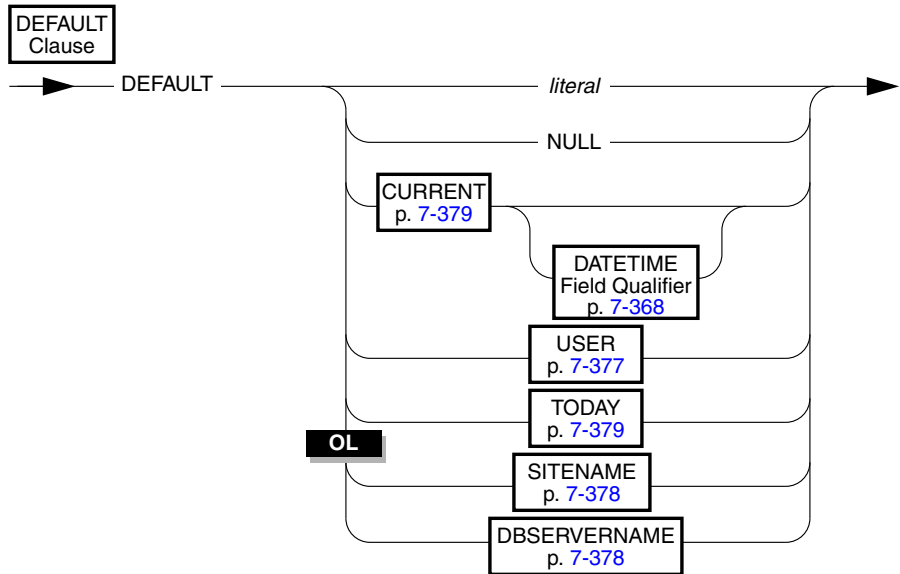
ADD Clause



column name is the name of an existing column before which the new column is to be placed.

new column name is the name of the column that you are adding.

Use the ADD clause to add a column to a table. You cannot add a SERIAL column to a table if the table has data in it.

DEFAULT Clause

literal represents a literal default.

DB

ESQL

The default value is inserted into the column when an explicit value is not specified. 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 use the keywords NOT NULL as part of the column definition.

You cannot place a default on SERIAL columns.

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

You can designate *literal terms* as default values. A literal term is a string of alpha or numeric constant characters defined by you. To use a literal term as a default value, follow these rules:

- Use integers with INTEGER, SMALLINT, DECIMAL, MONEY, FLOAT, and SMALLFLOAT columns.
- Use decimals with DECIMAL, MONEY, FLOAT, and SMALLFLOAT columns.
- Use characters with CHAR, VARCHAR, and DATE columns. Characters must be enclosed in quotation marks. Date literals must be of the format specified by the **DBDATE** environment variable. If **DBDATE** is not set, the format *mm/dd/yyyy* is assumed.
- Use literal INTERVAL values with INTERVAL columns. For information on using a literal INTERVAL, see [“Literal INTERVAL” on page 7-419](#).
- Use literal DATETIME values with DATETIME columns. For more information on using a literal DATETIME, see [“Literal DATETIME” on page 7-416](#).

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

Function Name	Data Type Requirements
CURRENT	DATETIME column with matching qualifier
DBSERVERNAME	CHAR or VARCHAR column at least 18 characters long
SITENAME	CHAR or VARCHAR column at least 18 characters long
TODAY	DATE column
USER	CHAR column at least 8 characters long

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

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

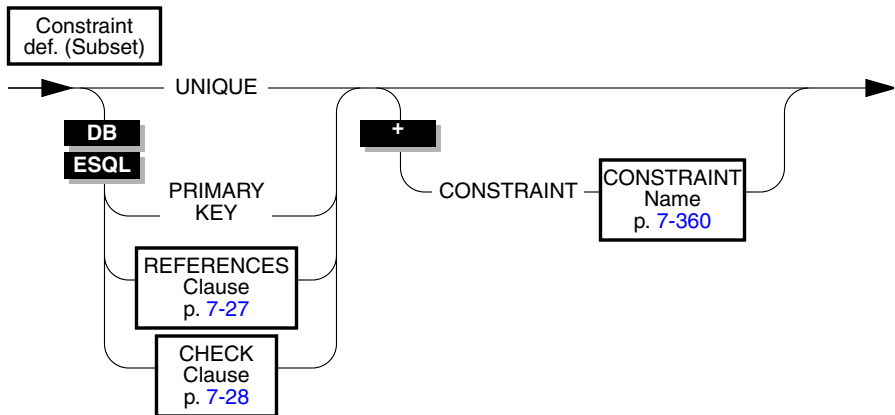
In this example, each *existing* row in the **items** table has a default value of 2.00 for the **item_weight** column. ♦

Using NOT NULL with ADD

If you do not indicate a default value for a column, the default is null *unless* you include the NOT NULL keywords after the data type of the column. In this case, if the NOT NULL keywords are used, there is no default value for the column and the column does not allow nulls. However, you cannot use the NOT NULL option when you add a column (unless both NOT NULL and a default value other than NULL is specified), nor can you specify that the new column has a unique or primary key constraint if the table contains data. If you want to add a column with a unique constraint, the table can contain a *single* row of data when you issue the ALTER TABLE statement. If you want to add a column with a NOT NULL or primary key constraint, the table *must* be empty when you issue the ALTER TABLE statement. The following statement is valid only if the **items** table is empty:

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

Subset of Constraint-Definition Option



You cannot specify a unique constraint on a new column if the table contains data. Nor can you have a unique constraint on a BYTE or TEXT column. If you want to add a column with a unique constraint, the table can contain a *single* row of data when you issue the ALTER TABLE statement.

I4GL
ISQL

DB

ESQL

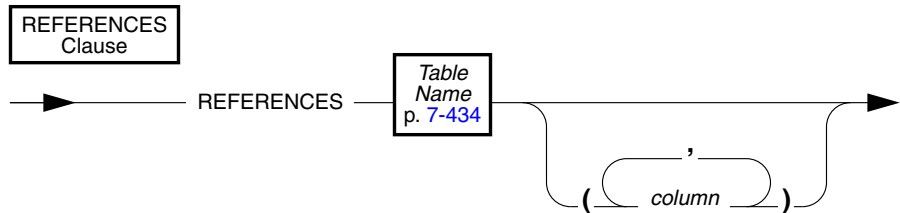
If you place a unique constraint on a column or set of columns, and there is already a unique index on that column or set of columns, the constraint shares the index. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before adding the unique constraint. ♦

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

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

- If you place a unique or primary key constraint on a column or set of columns, and there is already a unique index on that column or set of columns, the constraint shares the index. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before adding the constraint.
- If you place a unique or primary key constraint on a column or set of columns, and there is already a referential constraint on that column or set of columns, the duplicate index is upgraded to unique (if possible) and the index is shared.

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

REFERENCES Clause

DB

ESQL

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

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, where for every entry in the child (referencing) columns, there must exist a matching entry in the parent (referenced) columns.

The referenced column must be a column that either is part of a unique or primary key constraint. If the referenced column does not meet this criteria, the database server returns an error.

The foreign key (referencing column) can contain null and duplicate values, but every non-null value (that is, *all* foreign key columns contain non-null values) in the referencing columns *must* match a value in the referenced column.

A referential constraint has a one-to-one relationship between referencing and referenced columns. In other words, 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)
```

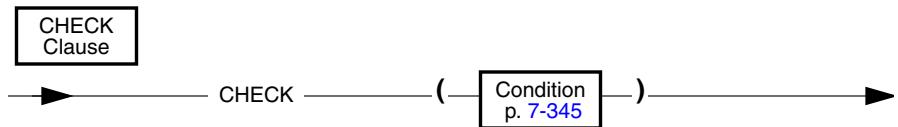
If you are referencing a primary key in another table, you do not have to explicitly state the primary key columns in that table. Referenced tables that do not specify the column to be referenced default to the primary key column. In the previous example, since **order_num** is the primary key in the **orders** table, you do not have to reference that column explicitly.

If you place a referential constraint on a column or set of columns, and there is already a duplicate or unique index 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 column is of type SERIAL. In this case, the primary key is of type SERIAL and the foreign key is of type INTEGER.

When a referential constraint is created, an exclusive lock is placed on the referenced table. The lock is released when the ALTER TABLE statement is done 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 of the check constraints defined on a table during an insert or update, the database server returns an error.

You cannot create check constraints for columns across tables. If 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 value to be entered is greater than 0.

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

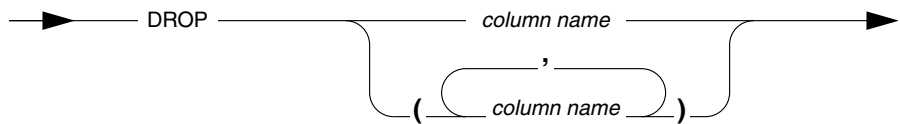


To create a constraint that checks values in more than one column, use the ADD CONSTRAINT clause. The next example adds the same column as the previous example. However, the check constraint now spans two columns in the table.

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



DROP Clause



column name is the name of the existing column that you wish to drop.

When you drop a column that is part of a multiple-column index, you drop the multiple-column index automatically. Similarly, when you drop a column that is part of a multiple-column constraint, you drop the multiple-column constraint automatically. ◆

When you drop a column, *all* constraints placed on that column are dropped, as follows:

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

Since any constraints associated with a column are dropped when the column is dropped, the structure of *other* tables may 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. This means that the structure of those other tables also have been altered. ◆

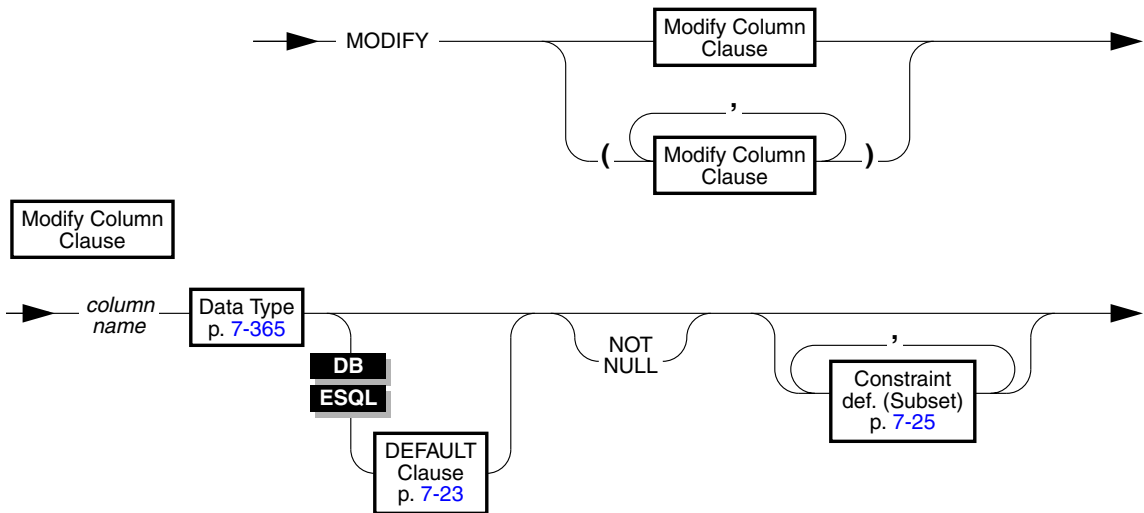
I4GL

ISQL

DB

ESQL

MODIFY Clause



column name is the name of the existing column that you wish to modify.

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

When you modify a column that has *single-column constraints* associated with it, *all* those constraints (in this case, the NOT NULL and UNIQUE attributes) are dropped. If you want certain attributes of the column to remain, you must specify them again. For example, if you are changing the data type of an existing column, **quantity**, to SMALLINT, and you want to continue disallowing nulls in this column, you can issue the following ALTER TABLE statement:

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

When you modify a column that is part of a multiple-column unique constraint, *all single-column constraints* are dropped but the multiple-column constraint is *not* dropped. For example, if you modify a column that does not allow nulls and is part of a multiple-column unique constraint, the NOT NULL constraint is dropped but the multiple-column constraint is not dropped. If you want the NOT NULL constraint to remain on the column, you must respecify NOT NULL in the MODIFY clause. ♦

I4GL

ISQL

DB

ESQL

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.

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. If you want certain attributes of the column to remain, you must respecify those attributes. For example, if you are changing the data type of an existing column, **quantity**, to SMALLINT, and you want to keep the default value (in this case, 1) and non-null attributes for that column, you can issue the following ALTER TABLE statement:

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

Note that both attributes are specified *again* in the MODIFY clause.

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

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

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

Using the MODIFY clause can 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. ♦

If 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))
```

I4GL

ISQL

DB

ESQL

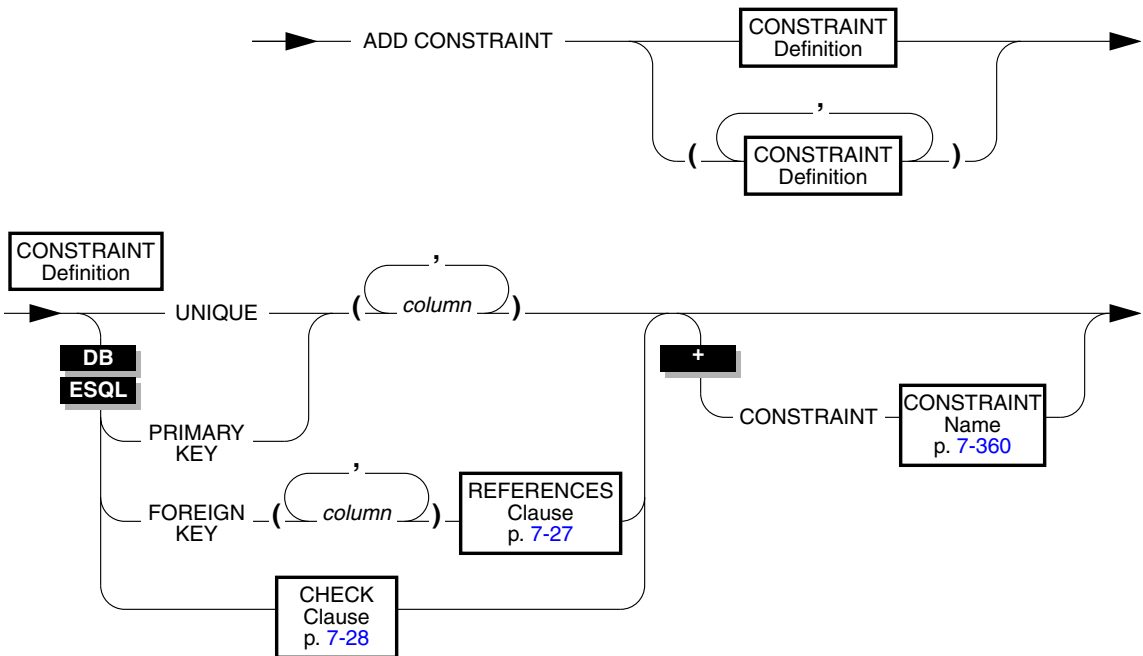
When there is a unique constraint, however, conversion takes place only if it does not violate the constraint. If a data conversion would result in duplicate values (by changing FLOAT to SMALLFLOAT, for example, or by truncating CHAR values), then the ALTER TABLE statement fails. ♦

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

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.

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 NOT NULL. However, if there is a unique index on the column, you should remove it using the DROP INDEX statement.

ADD CONSTRAINT Clause



column is the name of the column or columns on which the constraint is placed.

Use the `ALTER TABLE` statement with the `ADD CONSTRAINT` keywords to specify a constraint on a new or existing column or on a set of columns. For example, to add a unique constraint to the **fname** and **lname** columns of the **customer** table, use the following statement:

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

To name the constraint, change the example as follows:

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

I4GL

ISQL

SE

DB

ESQL

If 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. See the section [“SYSCONSTRAINTS” on page 2-16](#) for more information about the **sysconstraints** system catalog table.

The following rules apply when you add a unique constraint:

- The columns can contain only unique values.
- If you place a unique constraint on a column or set of columns, and there is already a unique index on that column or set of columns, the constraint shares the index. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before adding the unique constraint.
- An existing unique constraint cannot have the same name as the constraint you are adding.
- A composite list can include no more than 16 column names. The total length of all the columns cannot exceed 255 bytes. ♦
- A composite list can include no more than 8 column names and the total length of all the columns cannot exceed 120 bytes. ♦

The following rules apply when you add a unique or primary key constraint:

- If you place a unique or primary key constraint on a column or set of columns, and there is already a unique index on that column or set of columns, the constraint shares the index. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before adding the constraint.
- If you place a unique or primary key constraint on a column or set of columns, and there is already a referential constraint on that column or set of columns, the duplicate index is upgraded to unique (if possible) and the index is shared.
- If you place a referential constraint on a column or set of columns, and there is already a referential constraint on that column or set of columns, the duplicate index is upgraded to unique (if possible) and the index is shared.
- An existing unique constraint cannot have the same name as the constraint you are adding.
- A composite list can include no more than 16 column names. The total length of all the columns cannot exceed 255 bytes. ♦

14GL

ISQL

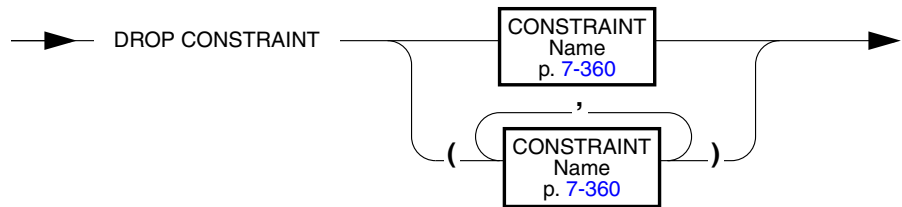
DB

ESQL

If you own the table or have Alter privilege on the table, you can create a unique constraint on the table and specify yourself as the owner of the constraint. If you have DBA privilege, you can create constraints for other users. ♦

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

DROP CONSTRAINT Clause



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

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

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

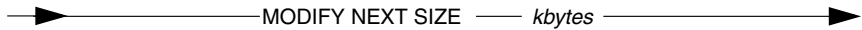
```
SELECT constrname FROM sysconstraints
WHERE tabid = (SELECT tabid FROM systables
WHERE tablename = "items")
```

DB

ESQL

If you drop a unique or primary key constraint that has a corresponding foreign key, those 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 also is dropped. ♦

MODIFY NEXT SIZE Clause



kbytes is the size, in kilobytes, that you want to assign for the next extent for this table.

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 following example:

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

The size of existing extents is not changed.

LOCK MODE Clause



Use the LOCK MODE keywords to change the locking mode of a table. The PAGE keyword is the default lock mode; it is set if the table is created without using the LOCK MODE clause. The following example sets the lock mode to row locking:

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

References

In this manual, see the following statements: CREATE TABLE, DROP TABLE, and LOCK TABLE.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of data integrity constraints and the discussion of creating a database and tables.

BEGIN WORK

Purpose

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

Syntax

```
+-----BEGIN WORK-----
```

Usage

The following code fragment shows how you might put statements within a transaction.

Figure 7-3

Code fragment showing statements within a transaction

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

Each row affected by an UPDATE, DELETE, or INSERT statement during a transaction is locked and remains locked throughout the transaction. A transaction that contains a large number of such statements, or that contains statements affecting a large number of rows, can exceed the limits placed by your operating system or IBM Informix OnLine configuration 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. As with all locks, this table lock is released when the transaction terminates.

ESQL

If you issue a `BEGIN WORK` statement while you are in a transaction, the database server returns an error.

You only can issue the `BEGIN WORK` statement if a transaction is not in progress.

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. This prevents the program from looping if the `ROLLBACK WORK` statement encounters an error or a warning. ♦

With ANSI-Compliant Databases

ANSI

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

- `DATABASE`
- `COMMIT WORK`
- `CREATE DATABASE`
- `ROLLBACK WORK`
- `START DATABASE`

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

References

In this manual, see the following statements: `COMMIT WORK` and `ROLLBACK WORK`.

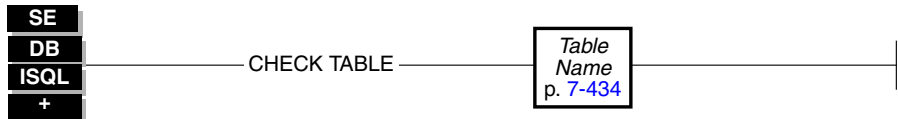
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of transactions and locking.

CHECK TABLE

Purpose

Use the CHECK TABLE statement to compare the data in a table with its indexes to determine whether they match. Use this statement when you think the data or the indexes might be corrupted because of a power failure, computer crash, or other abnormal program stoppage.

Syntax



Usage

Specify the name of the database table for which you want to check the data and associated indexes. For example:

```
CHECK TABLE cust_calls
```

The CHECK TABLE statement calls the **bcheck** utility. See the *IBM Informix SE Administrator's Guide* for a full description of the **bcheck** utility.

You cannot use the CHECK TABLE statement on a table unless you own it or have DBA status.

You cannot use the CHECK TABLE statement on the system catalog table **sysstables**, as it is always open. Instead, you can run the **bcheck** utility from the operating system prompt. You cannot use the CHECK TABLE statement on other system catalog tables unless you are user **informix**.

References

In this manual, see the REPAIR TABLE statement.

In the *IBM Informix SE Administrator's Guide*, see the discussion of the **bcheck** utility.

CLOSE

Purpose

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

Syntax

14GL	—	CLOSE	—	<i>cursor name</i>	—	
ESQL						

cursor name is the name of a cursor that has been declared with a DECLARE statement.

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 cursor that is associated with an INSERT statement is treated differently by a CLOSE statement than one associated with a SELECT statement.

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

An error code is returned if you close a cursor that was not open. No other action occurs. ♦

ANSI

Closing a SELECT Cursor

When *cursor name* is associated with a SELECT statement, closing the cursor terminates the SELECT statement. The database server releases all resources it may 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 may have been holding on rows selected through the cursor. If the CLOSE statement is contained in a transaction, the locks are not released by the database server until a COMMIT WORK or ROLLBACK WORK is executed.

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

Closing an INSERT Cursor

When *cursor name* 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 in the SQLCA structure, the product-specific name of which is shown in the following chart. (For information on using SQLERRD to count the total number of rows inserted, see the PUT statement on page 7-230.)

4GL	ESQL/C	ESQL/COBOL
SQLCA.SQLERRD[3]	sqlca.sqlerrd[2]	SQLERRD(3) OF SQLCA

The SQLCODE field of the SQLCA structure 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, SQLCODE is set to a negative error message number. See the following chart for the field name for each product.

4GL	ESQL/C	ESQL/COBOL
STATUS SQLCA.SQLCODE	sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA

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 have reopened it.

If the insert is not successful, the number of successfully inserted rows is stored in SQLERRD. Any buffered rows following the last successfully inserted row are discarded. Since in this case the CLOSE statement failed, the cursor is not closed. A second CLOSE statement can be successful because there are no longer any buffered rows. A subsequent OPEN statement also should be successful because the OPEN statement performs an implicit close that will succeed. An example of a situation in which a CLOSE statement fails and produces this setting is if there is insufficient disk space available for some of the rows to be inserted.

Using End of Transaction to Close a Cursor

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

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

For the use of insert cursors and the WITH HOLD clause, see the DECLARE statement on page [7-107](#).

References

In this manual, see the following statements: DECLARE, FETCH, FLUSH, FREE, OPEN, and PUT.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of cursors.

CLOSE DATABASE

Purpose

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.

You also can use the START DATABASE and ROLLFORWARD DATABASE statements after CLOSE DATABASE. ♦

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

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

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

```
DATABASE stores5
.
.
.
CLOSE DATABASE
DROP DATABASE stores5
```

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

SE

I4GL

ESQL

ESQL

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 prevents the program from looping if the `ROLLBACK WORK` statement encounters an error or a warning. ♦

References

In this manual, see the following statements: `CREATE DATABASE`, `DATABASE`, and `DROP DATABASE`.

COMMIT WORK

Purpose

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

Syntax

COMMIT WORK _____

Usage

Use the COMMIT WORK statement when you are sure you want to keep changes made to the database since the beginning of a transaction.

The COMMIT WORK statement releases all row and table locks.

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

Do not use the COMMIT WORK statement within a FOREACH loop, because it closes all open cursors except those declared with hold. ♦

References

In this manual, see the following statements: BEGIN WORK, ROLLBACK WORK, and DECLARE.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of transactions.

I4GL

ESQL

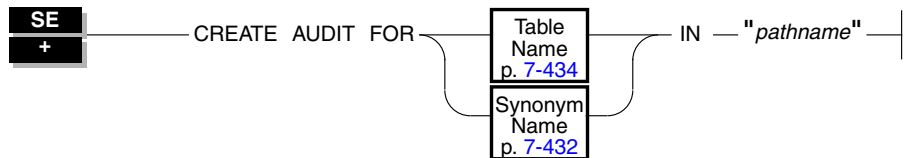
I4GL

CREATE AUDIT

Purpose

Use the CREATE AUDIT statement to create an audit trail file and to start writing the audit trail for an IBM Informix SE database.

Syntax



pathname specifies the full operating system pathname and file for the audit trail file.

Usage

You can create an audit trail to keep a record of all modifications to a table. An audit trail is a complete history of all additions, deletions, and updates to the table. The audit trail is used to reconstruct the table from a backup copy made at the time the audit trail is created.

You only can use the CREATE AUDIT statement with IBM Informix SE. IBM Informix OnLine provides for full database logging using log files.

You must own the *table or view* or have DBA status to use the CREATE AUDIT statement. You must set Execute privilege for all directories below **root** in *pathname* for each class of user (owner, owner's group, and public) that accesses your database.

If an audit trail file with the same *pathname* already exists, the CREATE AUDIT statement does nothing. If an audit trail file for the same table exists with a different *pathname*, an error message is returned.

Make a backup copy of your database files as soon as you run the CREATE AUDIT statement, but before you make any further changes to the database. (See the RECOVER TABLE statement for an example.) If possible, put the audit trail file on a different physical device from the one that holds your data, so that a failure of one does not damage the data on the other.

Audit trails slow your access to the database very slightly; each alteration of the database is recorded in the audit trail file, as well as in the database files.

The following example shows how to use the CREATE AUDIT statement in a UNIX environment:

```
CREATE AUDIT FOR orders IN "/u/safe/orders.aud"
```

References

In this manual, see the following statements: DROP AUDIT and RECOVER TABLE.

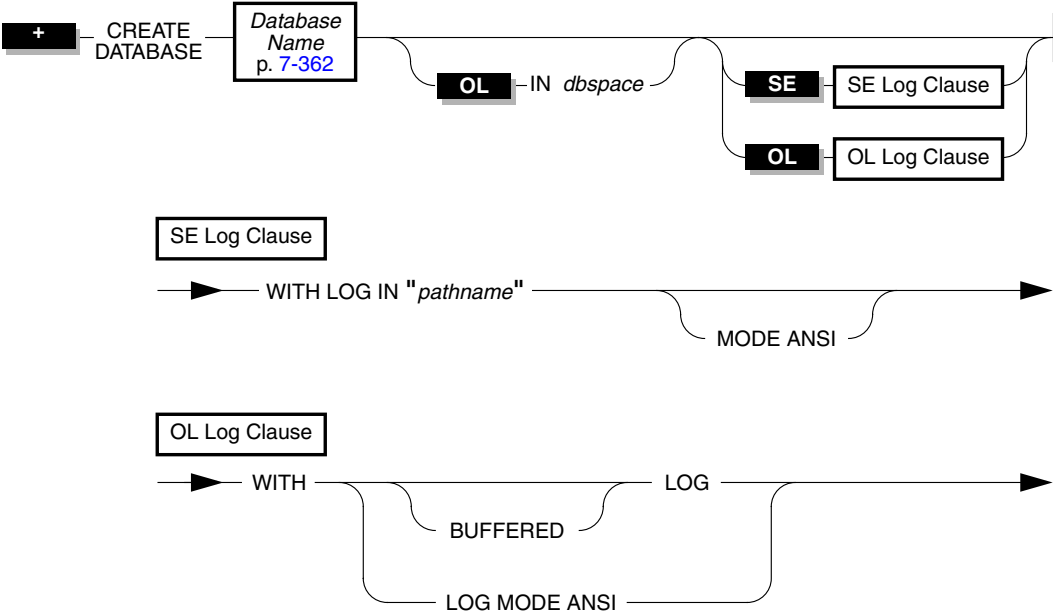
For more information on audit trails, see the manual for your application development tool.

CREATE DATABASE

Purpose

Use the CREATE DATABASE statement to create a new database.

Syntax



dbspace is the name of the dbspace in which you want to store the data for this database. The dbspace must already exist.

pathname is the full pathname, including the filename, for the log file.

Usage

The database that you create becomes the current database.

The database name you use must be unique within the IBM Informix OnLine system on which you are working. IBM Informix OnLine creates the system catalog tables containing the data dictionary that describes the structure of the database in the dbspace. If you do not specify the dbspace, IBM Informix OnLine creates the system catalog tables in the root dbspace.

The following statement creates the **stores5** database in the root dbspace:

```
CREATE DATABASE stores5
```

The following statement creates the **stores5** database in the **research** dbspace:

```
CREATE DATABASE stores5 IN research
```

The following example creates the **stores5** database in your current directory:

```
CREATE DATABASE stores5
```

The data for the database is placed in a subdirectory of your current directory with the name *database-name.dbs*. The system catalog, tables, data, and index files are placed in this directory, except for tables that you explicitly instruct be placed elsewhere (see the CREATE TABLE statement on page 7-75). The rules for directory names on your operating system govern the length of the name that you choose for the database. ♦

In IBM Informix 4GL and the embedded products, the CREATE DATABASE statement cannot appear in a multistatement PREPARE operation. ♦

SE

I4GL

ESQL

ANSI-Compliant Databases

You have the option of creating an ANSI-compliant database.

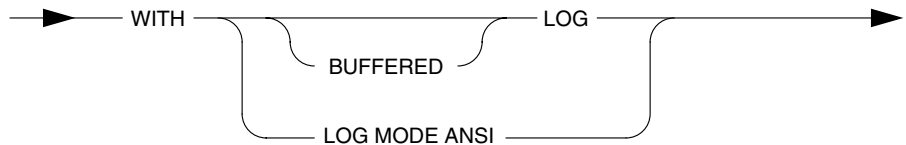
ANSI-compliant databases are set apart from non-ANSI databases by the following features:

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

There are other slight differences between databases that are and are not ANSI-compliant. These differences are noted as appropriate with the related SQL statement. ♦

Logging on IBM Informix OnLine

Use the following syntax to start logging transactions to the database on an IBM Informix OnLine system.



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

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

Designating Buffered Logging

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

```
CREATE DATABASE stores5 WITH BUFFERED LOG
```

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

ANSI

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

Designating an ANSI-Compliant Database

The following example creates an ANSI-compliant database:

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

Logging on IBM Informix SE

SE

Use the following syntax to start a log file for an IBM Informix SE database:

```
WITH LOG IN "pathname" 
```

pathname assigns the pathname of the log. It must be 64 or fewer characters long.

The following example creates an IBM Informix SE database named **accounts** with a log file. You must use the full pathname to designate the log file.

```
CREATE DATABASE accounts WITH LOG IN "/acct/f1990/acct_log"
```

If you specify the WITH LOG IN keywords, you can use transactions and the statements associated with transactions (BEGIN WORK, COMMIT WORK, and ROLLBACK WORK). Conversely, if you do not specify the WITH LOG IN keywords, you cannot use transactions.

You can use the START DATABASE statement to assign a log file to an existing IBM Informix SE database or to assign a new log file with a different name.

You can determine the location of the log file for the current database by running the following SELECT statement:

```
SELECT dirpath FROM informix.systables
WHERE tabtype = "L"
```

◆

Designating an ANSI-Compliant IBM Informix SE Database

The following example creates an ANSI-compliant database:

```
CREATE DATABASE employees WITH LOG IN "/u/acctg/lfile" MODE ANSI
```

◆

SE

References

In this manual, see the following statements: CLOSE DATABASE, DATABASE, DROP DATABASE, and START DATABASE.

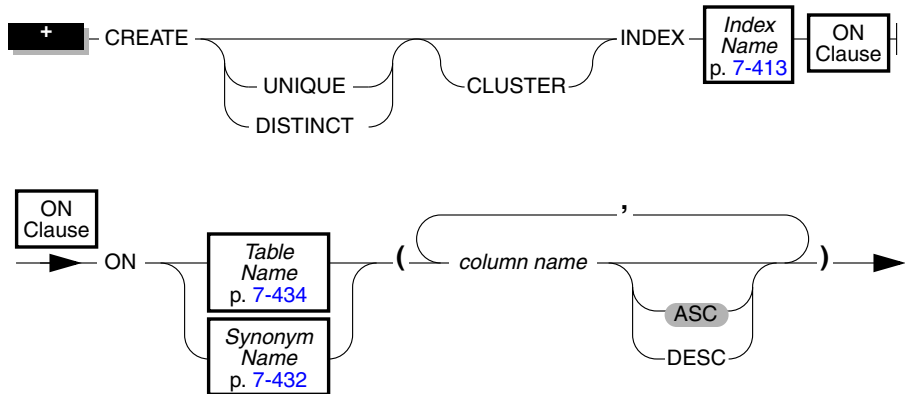
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of creating a database.

CREATE INDEX

Purpose

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.

Syntax



column name is the name of a column you want to index. To create an index that applies to several columns, enter a list of column names, separated by commas. All the columns you specify must belong to the same table.

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.

Only one index on a particular sequence of columns is allowed with the same ascending or descending order.

You cannot use a ROLLBACK WORK statement to undo a CREATE INDEX statement. If you roll back a transaction that contains a CREATE INDEX statement, the index remains and you do not receive an error message. ♦

UNIQUE Option

The following example creates a unique index:

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

This index prevents duplicates in the **customer_num** column. A column with a unique index can have, at most, one null value. The DISTINCT keyword is a synonym for the keyword UNIQUE, so the following statement would accomplish the same task:

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

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

You also can prevent duplicates in a column or set of columns by creating a unique constraint with the CREATE TABLE or ALTER TABLE statement. See the CREATE TABLE or ALTER TABLE syntax for more information.

CLUSTER Option

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

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

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

SE

You cannot create a CLUSTER index on a table that has an audit trail. ♦

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 up to 16 columns in a composite index. All columns indexed in a single CREATE INDEX statement cannot exceed 255 bytes.

SE

You can use up to 8 columns in a composite index. All columns indexed in a single CREATE INDEX statement cannot exceed 120 bytes. ♦

The ASC and DESC Keywords

Use the ASC option to specify an index that is maintained in ascending order. The ASC option is the default ordering scheme. Use the DESC option to specify an index that is maintained in descending order. When a column or list of columns is defined as unique in a CREATE TABLE or ALTER TABLE statement, the database server implements that UNIQUE CONSTRAINT by creating a unique ascending index. Thus, you cannot use the CREATE INDEX statement to add an ascending index to a column or column list 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 cathtmp ON customer (customer_num DESC)  
CREATE INDEX c_temp2 ON customer (customer_num, zipcode)
```

References

In this manual, see the following statements: ALTER INDEX, DROP INDEX, and CREATE TABLE.

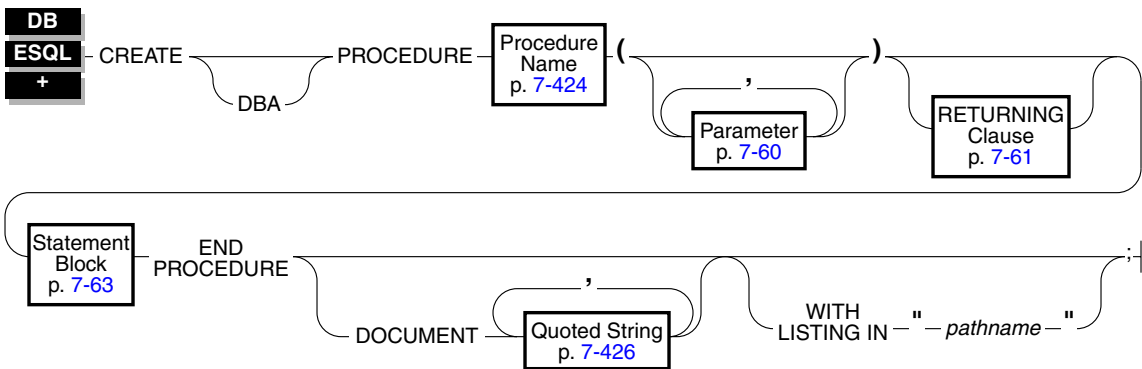
In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of indexes.

CREATE PROCEDURE

Purpose

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

Syntax



pathname is the full pathname of the file which is to contain the warnings of the procedure. The file must be on the host machine of the database server that serves the database.

Usage

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

You only can use a CREATE PROCEDURE statement 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. ♦

ESQL

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

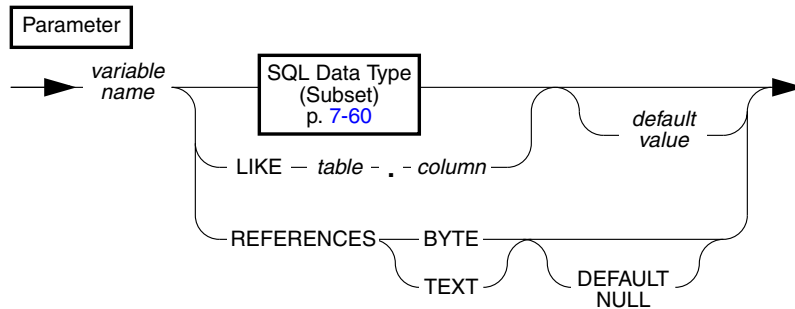
You must have at least Resource privilege on a database to create a procedure.

You cannot use a ROLLBACK WORK statement to undo a CREATE PROCEDURE statement. If you roll back a transaction that contains a CREATE PROCEDURE statement, the procedure remains and you do not receive an error message. ♦

DBA Option

If you create a procedure using the DBA option, the procedure 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. See the section [“Privileges on Stored Procedures” on page 8-16](#) for more information.

Parameter Syntax and Use



column is the name of the column of the type the variable is to be.

default value is the default value for the parameter.

table is the name of the table that contains *column*.

variable name is the name of a parameter used in the procedure.

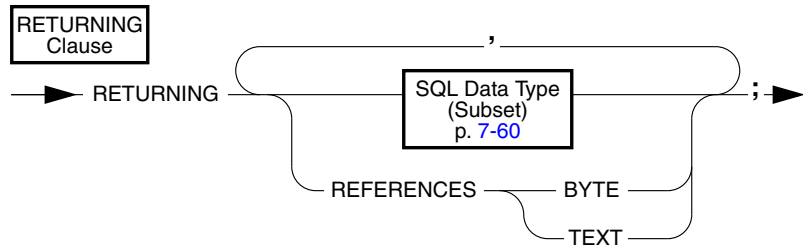
If you provide a default value for a parameter, that value is used if the procedure is called with less than the necessary arguments.

Subset of SQL Data Types Allowed in the Parameter List

The SQL Data Type subset includes all of the SQL data types except SERIAL, TEXT, and BYTE. For the complete syntax of all of the SQL data types, see page [7-365](#).

To use a TEXT or BYTE type, use the REFERENCES keyword, as shown in the diagram on page [7-60](#).

RETURNING Clause



A procedure can return zero or more values or sets of values. A procedure that returns more than one set of values (such as multiple rows from a table) is a *cursor* procedure. For example, the first RETURNING clause shown in the following example can return zero or one value if it is not a *cursor* procedure; if it is *cursor*, it returns more than one row from a table and each returned row contains one value. The second RETURNING clause can return zero or two values; if it is *cursor*, it returns more than one row with two values returned for each row.

```
RETURNING INT;

RETURNING INT, INT;
```

The receiving procedure or program must be written appropriately to accept the information.

Describing the Procedure in the DOCUMENT Clause

The quoted string or strings in the DOCUMENT clause should 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 of the procedures stored in the database.

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

```
SELECT data FROM sysprocbody b, sysprocedures p
WHERE b.procid = p.procid {join between the two catalogs}
AND
  p.procname = "do_something" {look for procedure do_something}
  AND b.datakey = 'D' { want user document }
ORDER BY b.seqno; { ... in order }
```

The rows returned are the complete text as supplied in the DOCUMENT clause of the CREATE PROCEDURE statement.

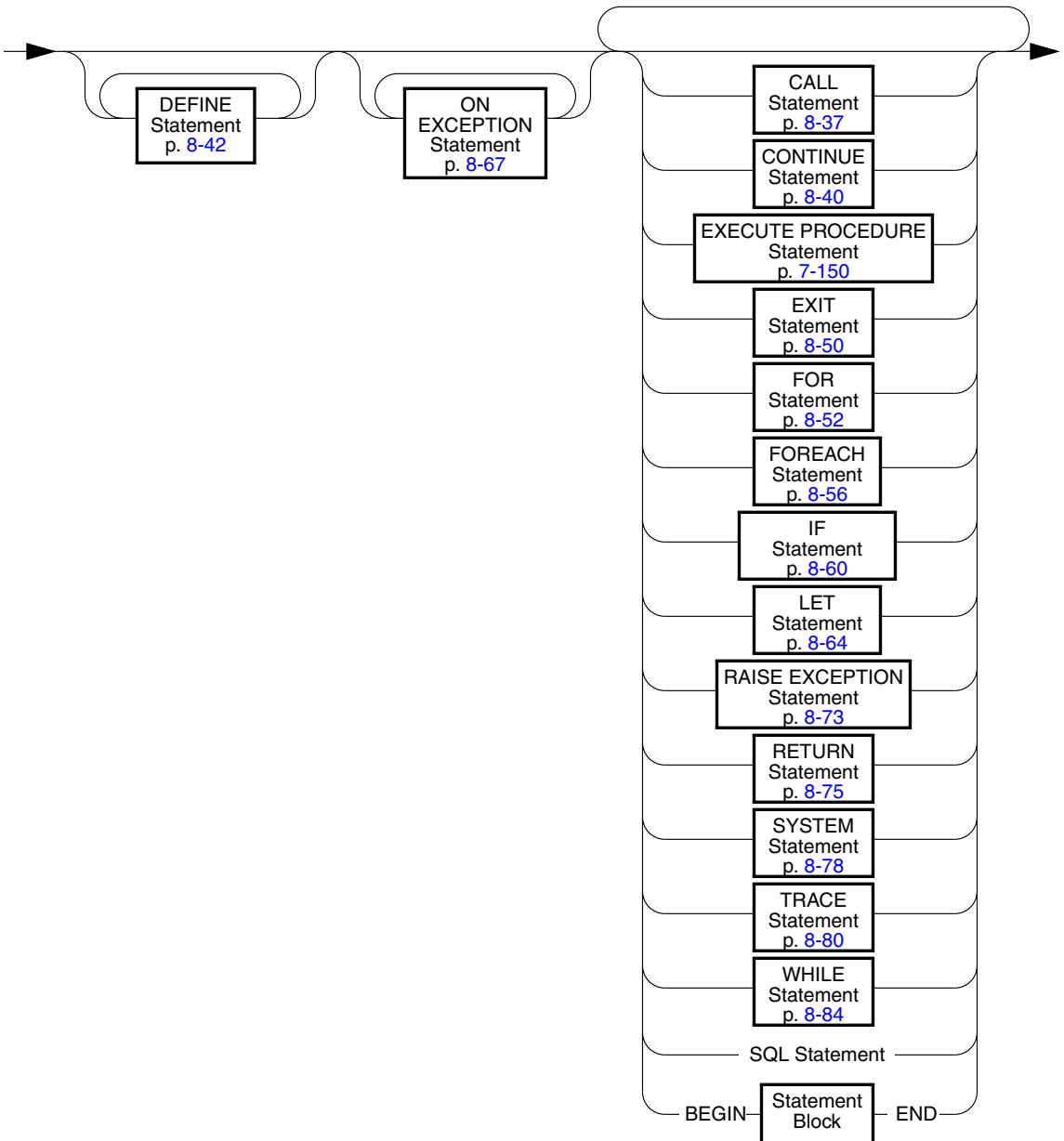
```
CREATE PROCEDURE ret_sal (dep_no INT, name CHAR(8) default user)
    RETURNING INT;
.
.
.
END PROCEDURE
DOCUMENT
    "Usage: salary (dept [required], name [default: your name])",
    "returns your (or someone else's) salary",
    "Warning: This procedure sends mail on illegal use"
WITH LISTING IN "/usr/tmp/sal.warnings";
```

WITH LISTING IN Option

The WITH LISTING IN option specifies a filename where compile time warnings are sent. After a procedure is compiled, this file holds one or more warning messages.

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

The Statement Block



The statement block can be empty, which results in a procedure that does nothing.

You cannot close the current database or select a new database within a procedure.

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 for those listed in [Figure 7-4](#).

Figure 7-4

SQL statements that cannot be used in a stored procedure

CHECK
CLOSE DATABASE
CREATE DATABASE
CREATE PROCEDURE
CREATE PROCEDURE FROM
DATABASE
INFO
LOAD
OUTPUT
REPAIR
ROLL FORWARD DATABASE
START DATABASE
UNLOAD

You can use a SELECT statement only in 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.

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 of the statements listed in [Figure 7-5](#). This ensures that changes cannot be made that affect the SQL statement that contains the procedure call.

Figure 7-5

SQL statements not allowed in a procedure that is called in a data manipulation statement

ALTER INDEX	DROP TABLE
ALTER TABLE	DROP VIEW
BEGIN WORK	INSERT
COMMIT WORK	RENAME COLUMN
DELETE	RENAME TABLE
DROP INDEX	ROLLBACK WORK
DROP SYNONYM	UPDATE

For example, if you use the following INSERT statement, then 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 the preceding example, **dup_name** cannot execute the statements listed in [Figure 7-5](#). However, if **dup_name** is called within a statement that is not an INSERT, UPDATE, SELECT, or DELETE statement (namely EXECUTE PROCEDURE), **dup_name** can execute the statements listed in [Figure 7-5](#).

Note that you can use the BEGIN WORK and COMMIT WORK statements in procedures. You can start a transaction, finish a transaction, or start and finish a transaction in a procedure.

References

In this manual, see the following statements: DROP PROCEDURE, GRANT, EXECUTE PROCEDURE, UPDATE STATISTICS, and REVOKE.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of creating and using stored procedures.

CREATE PROCEDURE FROM

Purpose

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

Syntax

```

ESQL
+
CREATE PROCEDURE FROM " filename "
ESQL
variable name

```

filename is the name of the file, or the name of the path and file, that contains the full text of the CREATE PROCEDURE statement.

variable name is a program variable that holds the name of the file that contains the full text of the CREATE PROCEDURE statement.

Usage

The *filename* provided in this statement is relative; if a simple filename is provided, the database server looks for the file in the current directory.

References

In this manual, see the CREATE PROCEDURE statement.

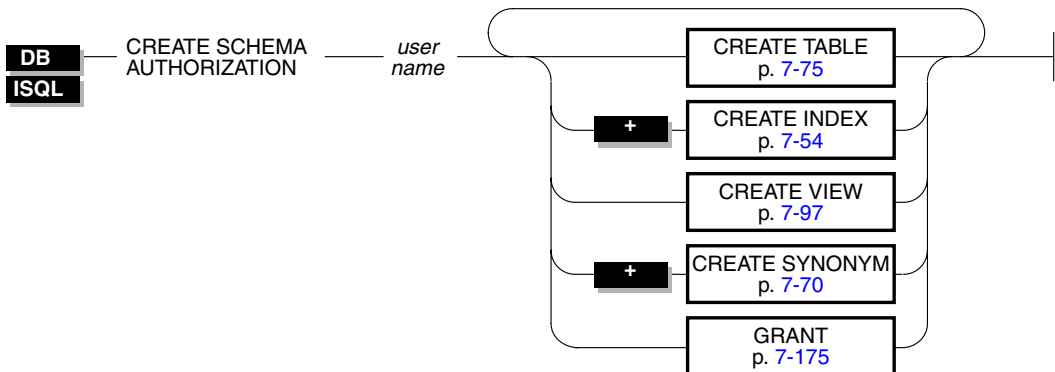
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of creating and using stored procedures.

CREATE SCHEMA

Purpose

The CREATE SCHEMA statement allows you to issue a block of CREATE and GRANT statements as a unit. It allows you to specify an owner/grantor of your choice for the subsequent block of CREATE and GRANT statements.

Syntax



user name is the name of the user to whom the CREATE SCHEMA statement block, and the objects created by the block, belongs.

Usage

You cannot issue the CREATE SCHEMA statement until the affected database has been created.

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

You can put CREATE and GRANT statements in any logical order within the statement, as shown in the following example. 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 > "12/31/1989"
CREATE INDEX idxtime ON mytable (mytime);
```

Creating Objects Within CREATE SCHEMA

All objects created by a CREATE SCHEMA statement are owned by *user name* even if you do not explicitly name each object. If you are the DBA, you can create objects for another user. If you are not the DBA and you attempt to create something for an owner other than *user name*, an error is returned.

Granting Privileges Within CREATE SCHEMA

You only can grant privileges using the CREATE SCHEMA statement; you cannot revoke or drop privileges.

Creating Objects or Granting Privileges Outside CREATE SCHEMA

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

References

In this manual, see the following statements: CREATE TABLE, CREATE INDEX, CREATE VIEW, CREATE SYNONYM, and GRANT.

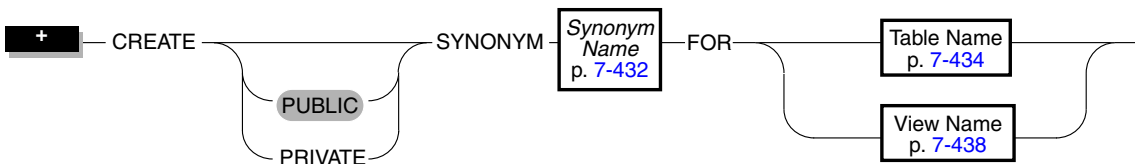
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of creating the data model.

CREATE SYNONYM

Purpose

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

Syntax



Usage

A public synonym is valid for all users, not just the creator. Users have the same privileges on a synonym that were granted to them on the table to which the synonym applies.

Once created, a synonym persists until the owner of the synonym 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 only for the lifetime of the SELECT statement. If a synonym refers to a table or view in the same database, the synonym is automatically dropped if the referenced table or view is dropped.

You cannot create a synonym for a synonym.

ANSI

The name of a synonym is qualified by the owner of the synonym (*owner.synonym*). The identifier *owner.synonym* must be unique among all the synonyms, tables, and views in the database. You must specify *owner* when you refer to a synonym owned by another user. The following example shows the convention:

```
CREATE SYNONYM emp FOR accting.employee
```



SE

You cannot use a ROLLBACK WORK statement to undo a CREATE SYNONYM statement. If you roll back a transaction that contains a CREATE SYNONYM statement, the synonym remains and you do not receive an error message. ◆

OL

You can create a synonym for any table or view on 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
```



STAR

You can create a synonym for any table or view that exists on any networked database server as well as on the database server which contains your current database. The database server that holds the table must be on-line when you create the synonym. IBM Informix STAR verifies that the object of the synonym exists when you create the synonym.

Here is an example of creating a synonym for an object that is not in the current database:

```
CREATE SYNONYM mysum FOR payables@phoenix:jean.summary
```

The identifier **mysum** now refers to the table **jean.summary**, which is in the **payables** database on the **phoenix** database server. Note that if the **summary** table is dropped from the **payables** database, the **mysum** synonym is left intact. Subsequent attempts to use **mysum** return a "Table not found" error. ◆

PUBLIC and PRIVATE Synonyms

If you use the PUBLIC keyword (or no keyword at all), your synonym can be used by anyone that has access to the database. If a synonym is public, a user does not need to know the name of the owner of the synonym. Any synonym in a database that is not ANSI-compliant *and* was created before the Version 5.0 of the database server is a public synonym.

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 only can be used by the owner of the synonym or if the owner's name is specified explicitly with the synonym. There can be more than one private synonym with the same name in the same database. However, each synonym with that name must be owned by a different user.

You only can own 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, and you use a synonym by its unqualified name, the private synonym is used.

If you use DROP SYNONYM with a synonym, and there are multiple synonyms with the same name, the private synonym is dropped. If you issue the DROP SYNONYM statement again, the public synonym is dropped.

OL

STAR

Chaining Synonyms with IBM Informix OnLine and IBM Informix STAR

If you create a synonym for a table that is not in the current database, and the base table for the synonym is dropped, the synonym stays in place. You can then create a new synonym for the dropped table, with the name of the dropped table as the synonym name, pointing 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 up to 16 synonyms in this manner.) ♦

The following steps chain two synonyms together for the **customer** table, which will ultimately reside on the **zoo** database server. (Note that the CREATE TABLE statements are not complete.)

1. In the **stores5** database on the database server called **training**:

```
CREATE TABLE customer (lname CHAR(15)...) 
```

2. On the database server called **acctng**:

```
CREATE SYNONYM cust FOR stores5@training.customer 
```

3. On the database server called **zoo**:

```
CREATE TABLE customer (lname CHAR(15)...) 
```

4. On the database server called **training**:

```
DROP TABLE customer  
CREATE SYNONYM customer FOR stores5@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**:

```
CREATE TABLE customer (lname CHAR(15)...) 
```

2. On the database server called **acctng**:

```
CREATE SYNONYM cust FOR stores5@training.customer 
```

3. On the database server called **training**:

```
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

In this manual, see the DROP SYNONYM statement.

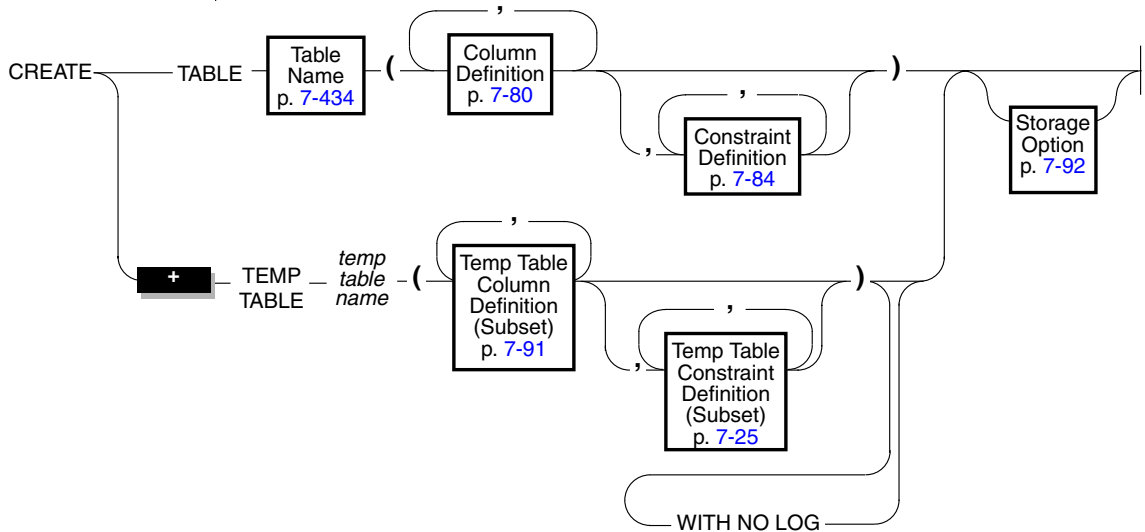
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of synonyms.

CREATE TABLE

Purpose

Use the CREATE TABLE statement to create a new table in the current database, place data integrity constraints on its columns or on a combination of its columns, designate the size of its initial and subsequent extents, and specify how it is to be locked.

Syntax



temp table name

is the name that you want to assign to the temporary table. You cannot use the *owner.* convention.

ANSI

Usage

Table names must be unique in the same 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 an ANSI-compliant database, the combination *owner.tablename* must be unique within the database. ♦

By default, all users who have been granted Connect privilege to a database have all access privileges (except Alter, Index, 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 you want to give to specific users.

ANSI

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

SE

You cannot use a ROLLBACK WORK statement to undo a CREATE TABLE statement. If you roll back a transaction that contains a CREATE TABLE statement, the table remains and you do not receive an error message. ♦

ISQL

Using the CREATE TABLE statement outside the CREATE SCHEMA statement generates warnings if you use the **-ansi** flag or set DBANSIWARN. ♦

DB

I4GL

Using the CREATE TABLE statement generates warnings if you use the **-ansi** flag or set DBANSIWARN. ♦

ESQL

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 also can implement data integrity constraints and turn effective checking off and on. For information on data integrity constraints, see the *IBM Informix Guide to SQL: Tutorial*. For information on effective checking, see the SET CONSTRAINTS statement on page 7-289.

Tip: In a database without logging, the only constraint-checking mode available is detached. 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 only can apply 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.

Limits on Constraint Definitions

You can include up to 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 can use up to 8 columns in an IBM Informix SE list of columns. The total length of all columns cannot exceed 120 bytes. ♦

Adding or Dropping Constraints

Once 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.

SE

I4GL

ISQL

DB

ESQL

Enforcing Primary Key, Unique, and Referential Constraints

Unique constraints are *implemented* as an ascending index that allows only unique entries. After one of these constraints is placed on a column, the database server creates a unique index for the unique constraint.

Since constraints are enforced through indexes, you cannot create an index (using the CREATE INDEX statement) for a column that is of the same type as the constraint placed on that column. For example, if there is a unique constraint on a column, you can create neither a unique index for that column nor a duplicate ascending index. ♦

Primary key, unique, and referential constraints are *implemented* either 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, an index is not built. Instead, the 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.

Since these constraints are enforced through indexes, you cannot create an index (using the CREATE INDEX statement) for a column that is of the same type as the constraint placed on that column. For example, if there is a unique constraint 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 unique constraint. The index name in the **sysindexes** system catalog table is created with the following format:

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

where *tabid* and *constraint id* are from the **sysables** and **sysconstraints** system catalog tables, respectively.

I4GL

ISQL

DB

ESQL

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>

where *constraint type* is the letter **u** (for unique constraint). If the name conflicts with an existing identifier, the database server returns an error and you must then supply a *constraint name*. ♦

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>*

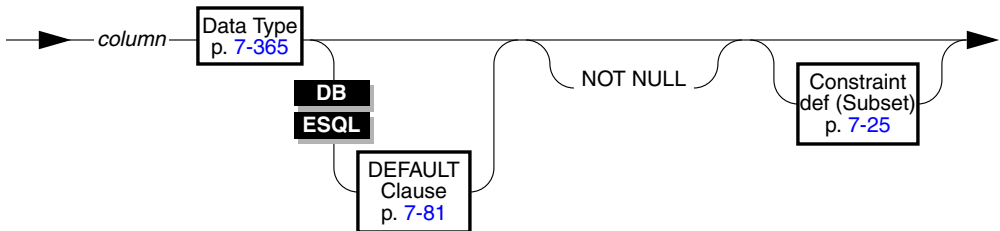
where *tabid* and *constraint id* are from the **systables** and **sysconstraints** system catalog tables, respectively.

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>

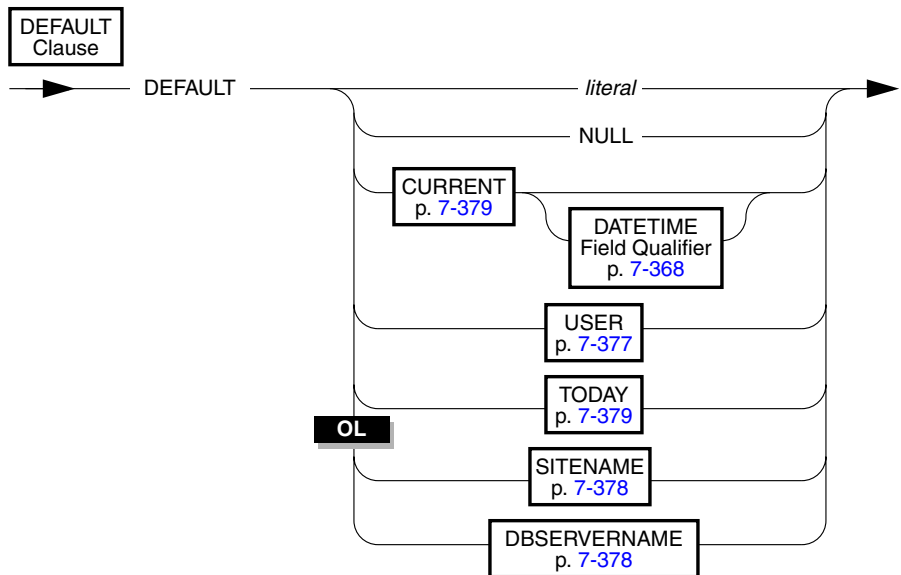
where *constraint type* is the letter **u** for unique or primary key constraints, **r** for referential constraints, or **c** for check constraints. If the name conflicts with an existing identifier, the database server returns an error and you must then supply a *constraint name*. ♦

Column-Definition Option



column is a valid identifier for columns. The column name must be unique within a table, but you can use the same names in different tables in the same database.

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.

The DEFAULT Clause

literal represents a literal default.

DB

ESQL

The default value is inserted into the column when an explicit value is not specified. 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 use the keywords NOT NULL as part of the column definition.

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

You can designate *literal terms* as default values. A literal term is a string of character or numeric constant characters defined by you. To use a literal term as a default value, follow these rules:

- Use integers with INTEGER, SMALLINT, DECIMAL, MONEY, FLOAT, and SMALLFLOAT columns.
- Use decimals with DECIMAL, MONEY, FLOAT, and SMALLFLOAT columns.

- Use characters with CHAR, VARCHAR, and DATE columns. Characters must be enclosed in quotation marks. Date literals must be of the format specified by the **DBDATE** environment variable. If **DBDATE** is not set, the format *mm/dd/yyyy* is assumed.
- Use literal INTERVAL values with INTERVAL columns. For information on using a literal INTERVAL, refer to [“Literal INTERVAL” on page 7-419](#).
- Use literal DATETIME values with DATETIME columns. For more information on using a literal DATETIME, refer to [“Literal DATETIME” on page 7-416](#).

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

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

Function Name	Data Type Requirement
CURRENT	DATETIME column with matching qualifier
DBSERVERNAME	CHAR or VARCHAR column at least 18 characters long
SITENAME	CHAR or VARCHAR column at least 18 characters long
TODAY	DATE column
USER	CHAR column at least 8 characters long

The next example creates a table **accounts**. In **accounts**, the **acc_type** and **acc_descr** columns have literal default values while **acc_id** 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)
◆
```

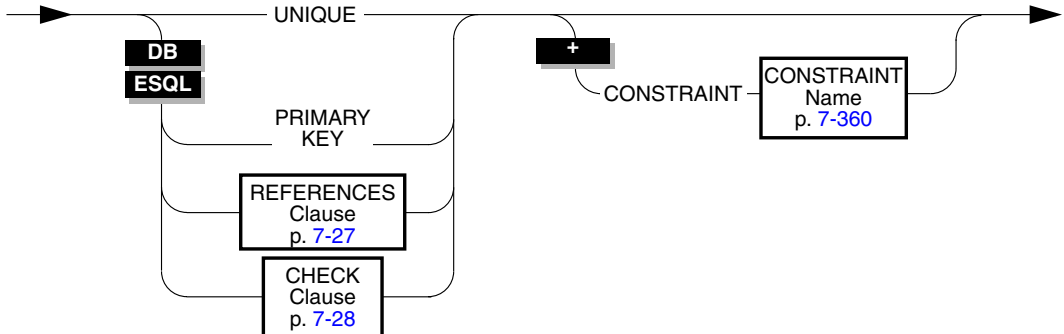

Specifying **NOT NULL** in a Column Definition

If you do not indicate a default value for a column, the default is null *unless* you include the NOT NULL keywords after the data type of the column. In this case, there is no default value for the column. The following example creates the **newitems** table. In **newitems**, the column **manu_code** does not have a default value nor does it allow nulls.

```
CREATE TABLE newitems (
    newitem_num INTEGER,
    manu_code CHAR(3) NOT NULL,
    promotype INTEGER,
    descrip CHAR(20))
```

If you designate a column as NOT NULL (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. See the Data Type segment on page 7-365 for more information.

Subset of Constraint-Definition Option



I4GL

ISQL

DB

ESQL

Unlike the table-level constraint-definition option, constraints at the column level are limited to a single column. In other words, you cannot create a unique multiple-column constraint. For more information on unique constraints see the “[The Constraint-Definition Option](#)” on page 7-84. ♦

Unlike the table-level constraint-definition option, constraints at the column level are limited to a single column. In other words, you cannot create check, unique, primary, or foreign key multiple-column constraints. For more information on the unique, primary key, and check constraints, see the “[The Constraint-Definition Option](#)” on page 7-84.

The following example creates a simple table with a unique and primary key and names the two constraints created:

```
CREATE TABLE accounts (
    acc_num INTEGER PRIMARY KEY CONSTRAINT num,
    acc_code UNIQUE CONSTRAINT code,
    acc_descr CHAR(30))
```

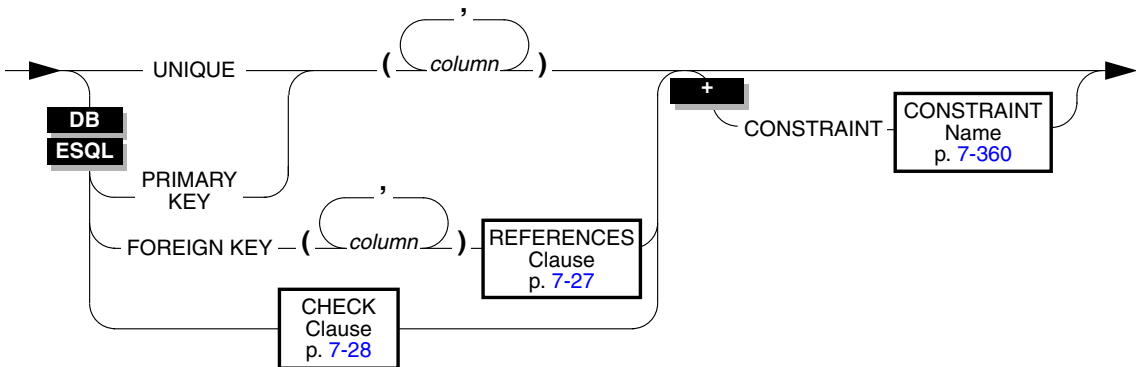
Using Blob Data Types in Constraints

You cannot have a unique constraint on a BYTE or TEXT column. ♦

You cannot place a unique, primary key, or referential constraint on BYTE or TEXT columns. However, you can check for null or non-null values by placing a check constraint on a BYTE or TEXT column. ♦

- I4GL
- ISQL
- DB
- ESQL

The Constraint-Definition Option



column is the name of the column.

The constraint-definition option allows you to create constraints for a single column or a set of columns. You can include up to 16 columns in a list of columns. The total length of all the columns cannot exceed 255 bytes.

You can use up to 8 columns in an IBM Informix SE list of columns. The total length of all the columns cannot exceed 120 bytes. ♦

- SE

Defining a Column as Unique

You can 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, you 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 all_name, a_code SERIAL)
```

You cannot place a unique constraint on a `BYTE` or `TEXT` column.

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 next 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))
```

You cannot place a primary key constraint on a `BYTE` or `TEXT` column. ♦

Defining a Column as a Foreign Key

A foreign key *joins* and establishes dependencies between tables. A foreign key references a unique or primary key in a table. For every entry in the foreign key columns, there must exist a matching entry in the unique or primary key columns if all foreign key columns contain non-null values. You cannot make `BYTE` or `TEXT` columns foreign keys. ♦

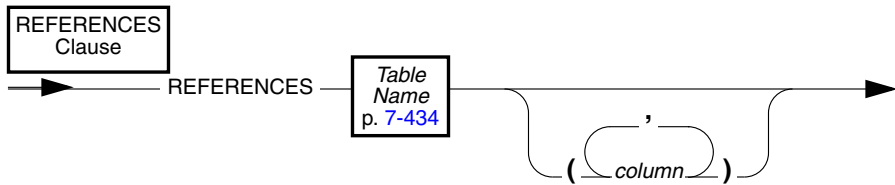
DB

ESQL

DB

ESQL

The REFERENCES Clause



DB

ESQL

You can use the REFERENCES clause to reference a column, or set of columns. If you are using the REFERENCES clause at the column level, you only can reference a single column. ♦

Referenced and Referencing Column Requirements

DB

ESQL

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 part of 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 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 columns (or primary key) and the child is the referencing columns (or 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 that manages that employee. In this case, **mgr_no** is the foreign key (or child) that references **emp_no**, the primary key (or 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 a 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 returns 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 columns to be referenced default to the primary key columns. The references section of the previous example can be rewritten as follows:

```
...  
    FOREIGN KEY (ref_num, ref_type) REFERENCES accounts  
    ...
```

Since **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. ♦

DB

ESQL

Data Type Restrictions

The data types of the referencing and referenced columns must be identical unless the column is of type SERIAL. In this case, the referenced column is type SERIAL, while the referencing column is type INTEGER. In the previous example, there is a one-to-one correspondence between the data types of the primary and foreign keys. If any one of these columns 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 a referential constraint is created, 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. ♦

DB

ESQL

Using REFERENCES in a Column Definition

When you use the references option at the column-definition level, you only 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))
```

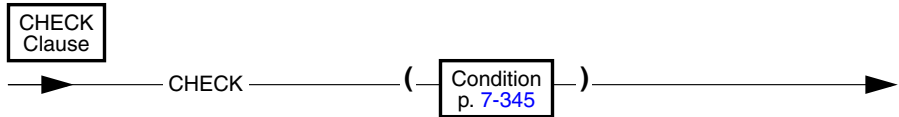
Note that **ref_num** is not explicitly called a foreign key in the column definition syntax. At the column level, the foreign key designation is applied automatically.

DB

ESQL

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 simply can reference the **accounts** table without specifying a column. Since **acc_num** is the primary key of the **accounts** table, it becomes, by default, the referenced column. ♦

The CHECK Clause



DB

ESQL

Check constraints allow you to designate conditions that must be met *before* data can be assigned to a column during an INSERT or UPDATE statement. If a row evaluates to *false* for any of the check constraints defined on a table during an insert or update, the database server returns an error.

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

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 next 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 type MONEY whose values must be between 0 and 99999. If, however, you wanted to test that both columns had 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. ♦

DB

ESQL

DB

ESQL

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. ♦

TEMP TABLE Option

Temporary tables last only for the duration of the program. If your database uses transactions and the temporary table was not created with the WITH NO LOG keywords, the temporary table is removed when you close your database.

DB

ISQL

A temporary table exists until you exit your application. If your database uses transactions and the temporary table was not created with the WITH NO LOG keywords, the temporary table is removed when you close your database.

The INFO statement and the Info Menu Option cannot be used with temporary tables. ♦

STAR

INET

Temporary tables are created on your local database server. (That is, in your current directory, if any, or in the database server process of your local machine.) If you create a temporary table and then close the database, the temporary table is deleted. However, if your current database is on your local database server and you close this database and open a remote database, your local database server temporary table is unavailable *until* you close the remote database and then reopen a local database. ♦

I4GL

ISQL

DB

ESQL

DB

ESQL

If you have Connect privilege on a database, you can create 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.

You cannot build a FORM4GL or PERFORM screen on a temporary table.

Subset of Column-Definition Option

You cannot place referential constraints on columns in a temporary table. In other words, temporary columns cannot be referenced or referencing columns. The following constraint-definition keywords cannot be used when you are creating a temporary table:

- REFERENCES
- CONSTRAINT

For more information on column constraint-definition options, refer to the [“Subset of Constraint-Definition Option” on page 7-25](#). ♦

Subset of Constraint-Definition Option

You cannot place referential constraints on columns in a temporary table. In other words, temporary columns cannot be referenced or referencing columns. The following table constraint-definition keywords cannot be used when you are creating a temporary table:

- FOREIGN KEY
- REFERENCES
- CONSTRAINT

For more information on table-level constraint-definition options, refer to [“The Constraint-Definition Option” on page 7-84](#). ♦

WITH NO LOG Option for Temporary Tables

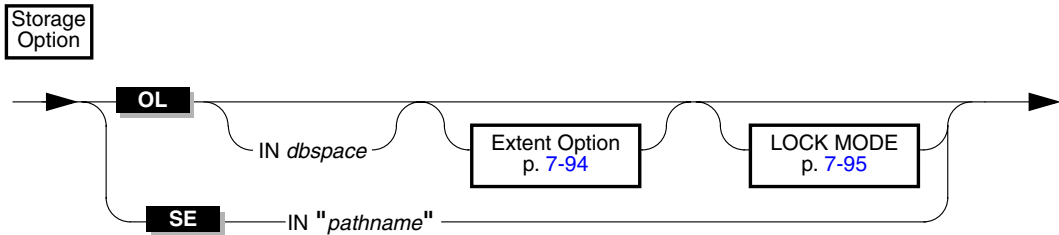
If you use the WITH NO LOG keywords in a CREATE TABLE statement and the database does not use logging, the WITH NO LOG option is ignored.

Once you turn off the 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 of temporary tables in a database that uses logging:

```
CREATE TEMP TABLE tab2 (fname CHAR(15), lname CHAR(15))  
WITH NO LOG
```

Storage Option



dbspace is the name of the dbspace in which the database table is to be stored.

pathname specifies the full operating system path and filename in which you want to store the database table, with no extension to the filename.

The storage option allows you to specify where the database table is stored and the locking granularity for the table.

The IN dbspace Clause

The dbspace that you specify must already exist and it must be on your local IBM Informix OnLine system. If you do not specify a dbspace, the default is the dbspace in which the current database resides.

The IN *dbspace* clause allows you to isolate a table if you need to do so. For example, if the **stores5** database is in the **stockdata** dbspace but you want the **customer** data to be in a separate dbspace called **custdata**, use the following statements.

Figure 7-6
Isolating a table in a separate dbspace

```

CREATE DATABASE stores5 IN stockdata

CREATE TABLE customer
(
  customer_num    SERIAL(101),
  fname           CHAR(15),
  lname           CHAR(15),
  company         CHAR(20),
  address1        CHAR(20),
  address2        CHAR(20),
  city            CHAR(15),
  state           CHAR(2),
  zipcode         CHAR(5),
  phone           CHAR(18)
)
IN custdata EXTENT SIZE 16

CREATE TABLE orders
(
  order_num       SERIAL(1001),
  order_date      DATE,
  customer_num    INTEGER,
  ship_instruct   CHAR(40),
  backlog         CHAR(1),
  po_num          CHAR(10),
  ship_date       DATE,
  ship_weight     DECIMAL(8,2),
  ship_charge     MONEY(6),
  paid_date       DATE
)
EXTENT SIZE 24 NEXT SIZE 12
.
.
.

```

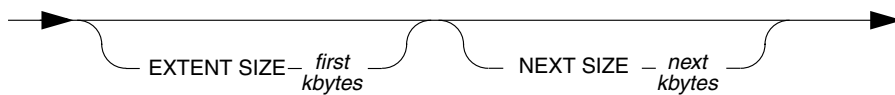
By default, the **orders** table is with the rest of the database, in **stockdata**.

If your table has one or more blob columns, you can store the blob data with the table data or in a separate blobspace. See [“Data Type” on page 7-365](#) for more information. The following example shows how blobspaces and dbspaces are specified.

This statement creates the **resume** table. The data for the table is stored in the **employ** dbspace. The data in the **resume** column is stored with the table, but the data associated with the **photo** column is stored in a blob space 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),
  resume         TEXT IN TABLE,
  photo          BYTE IN photo_space
)
IN employ
```

Extent Option



first kbytes is the length in kilobytes of the first extent for the table. The default size is eight kilobytes.

next kbytes is the length in kilobytes for the subsequent extents. The default size is eight kilobytes.

See the *IBM Informix Guide to SQL: Tutorial* for a discussion of how to calculate extent sizes.

The minimum size of an extent is four pages. If you specify an extent size (or next extent size) smaller than the minimum size, the database server returns an error.

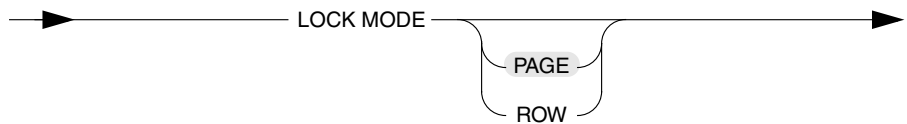
The following example specifies a first extent of 20 kilobytes and allows the rest of the extents to use the default size:

```
CREATE TABLE emp_info
(
  f_name CHAR(20),
  l_name CHAR(20),
  position CHAR(20),
  start_date DATETIME YEAR TO DAY,
  comments VARCHAR(255)
)
EXTENT SIZE 20
```

Revising Extent Sizes for Unloaded Tables

You can revise the CREATE TABLE statements in generated schema files to revise the extent and next extent sizes of unloaded tables. See the *IBM Informix OnLine Administrator's Guide* for more information.

LOCK MODE Clause



The default locking granularity is a page.

Row-level locking provides the highest level of concurrency. If you are using many rows at one time, the lock-management overhead can become significant. You also can exceed the maximum number of locks available, depending on the configuration of your OnLine system.

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

You can change the lock mode of an existing table with the ALTER TABLE statement.

The IN pathname Option

The *pathname* in an IN clause can specify any valid directory and is not restricted to the directory that contains the current database. This enables a database to include tables located on another host machine on the network.

In UNIX, the *pathname* cannot be longer than 64 characters and must be within quotes ("). A pathname is of the following form:

[/directory-name/...]filename

If the *pathname* in an IN clause specifies a filename that is different from the *table name*, always use the *table name* (rather than the filename) to refer to the table in subsequent SQL statements.

The creator of the table must have search permissions on all directories in the path and write permissions on the directory that is to contain the files. ♦

References

In this manual, see the following statements: ALTER TABLE, CREATE INDEX, CREATE DATABASE, and DROP TABLE. Also see the Data Type segment on page [7-365](#).

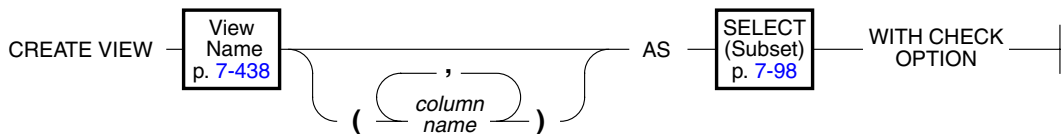
In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of data integrity, creating a table, and extent sizing.

CREATE VIEW

Purpose

Use the CREATE VIEW statement to create a new view based upon existing tables and views in the database.

Syntax



column name is an identifier that names a column of *view name*. The number of columns that you name must match the number of columns that you select.

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 INDEX	DROP TABLE
ALTER TABLE	LOCK TABLE
CREATE INDEX	RECOVER TABLE
CREATE TABLE	RENAME TABLE
DROP INDEX	UNLOCK TABLE

The view behaves like a table with the name *view name* and consists of the set of rows and columns returned by the *SELECT* statement each time the *SELECT* statement is executed by using the view. The view reflects changes to the underlying tables with one exception. If the view is defined with a *SELECT ** clause, it only has the columns in the underlying tables at the time the view is created. New columns added subsequently to the underlying tables using the *ALTER TABLE* statement do not appear in the view.

Data types of the columns of the view are inherited from the tables from which they come. Data types of virtual columns are determined from the nature of the expression.

You must have *Select* privilege on all columns from which the view is derived to create a view.

The *SELECT* statement is stored in the **sysviews** system catalog table. When you subsequently refer to a view in another statement, the database server performs the defining *SELECT* statement while it executes the new statement.

You cannot use a *ROLLBACK WORK* statement to undo a *CREATE VIEW* statement. If you roll back a transaction that contains a *CREATE VIEW* statement, the view remains and you do not receive an error message. ♦

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

Using the *CREATE VIEW* statement generates warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

Subset of a **SELECT** Allowed in **CREATE VIEW**

The *SELECT* statement is a statement of the form described on page [7-258](#), except that it cannot have an *ORDER BY* clause, *INTO TEMP* clause, or *UNION* operator. Do not use display labels in the select list; display labels are interpreted as column names.

SE

DB

ISQL

I4GL

ESQL

Naming View Columns

If you do not specify a list of columns for *view name*, the view inherits the column names of the underlying tables. In the following example, the view **herostock** has the same column names as in the SELECT statement:

```
CREATE VIEW herostock AS
  SELECT stock_num, description, unit_price, unit, unit_descr
  FROM stock WHERE manu_code = "HRO"
```

If the SELECT statement returns an expression, the corresponding column in the view is called a *virtual* column. You must provide a name for virtual columns. You also must 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 shown in the following example:

```
CREATE VIEW someorders (custnum,ocustnum,newprice) AS
  SELECT orders.order_num,items.order_num,
         items.total_price*1.5
  FROM orders, items
  WHERE orders.order_num = items.order_num
  AND items.total_price > 100.00
```

If you must provide names for some of the columns in a view, then you must provide names for all the columns; that is, the column list must contain an entry for every column appearing in the view.

Using a View in the SELECT Statement

You can define a view in terms of other views, except that you must abide by the restrictions on queries listed in the *IBM Informix Guide to SQL: Tutorial*.

WITH CHECK OPTION Keywords

The WITH CHECK OPTION keywords instruct the database server to ensure that all modifications to the underlying tables made through the view satisfy the definition of the view.

The following example creates a view named **palo_alto** that uses all of the information in the **customer** table for customers in the city of Palo Alto. The database server checks any modifications made to **customer** 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
```

Updating through Views

If a view is built on a single table, the view is said to be *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, created using an arithmetical expression

That is, in an updatable view, the values in the underlying table can be updated by inserting values into the view.

References

In this manual, see the following statements: CREATE TABLE, DROP VIEW, GRANT, and SELECT.

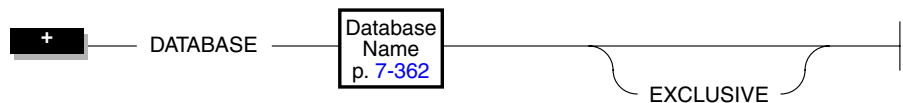
In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of views and security in Chapter 11.

DATABASE

Purpose

Use the DATABASE statement to select an accessible database as the current database.

Syntax



Usage

You can use the DATABASE statement to select any database on your IBM Informix OnLine database server. You can select a database on another OnLine database server by specifying the name of the database server with the database name.

The DATABASE statement closes any other current database, unless the current database is on another database server. In that case, you receive an error because you must explicitly close a database on another database server.

You cannot include the DATABASE statement in a multistatement PREPARE operation.

You can determine the type of database a user selects by checking the warning flag after a DATABASE statement in the SQLCA structure. See Chapter 5 of this manual for more information about the warning portion of the SQLCA structure.

I4GL

ESQL

If the database has transactions, the second element of the **sqlcawarn** structure contains a W after the DATABASE statement executes. See the following chart for the name of the variable used for each product.

4GL	ESQL/C	ESQL/COBOL
SQLCA.SQLAWARN[2]	sqlca.sqlwarn.sqlwarn1	SQLWARN1 OF SQLWARN OF SQLCA

◆

If the database is ANSI-compliant, the third element of the **sqlcawarn** structure contains a W after the DATABASE statement executes. See the following chart for the name of the variable used for each product.

4GL	ESQL/C	ESQL/COBOL
SQLCA.SQLAWARN[3]	sqlca.sqlwarn.sqlwarn2	SQLWARN2 OF SQLWARN OF SQLCA

◆

If the database is an IBM Informix OnLine database, the fourth element of the **sqlcawarn** structure contains a W after the DATABASE statement executes. See the following chart for the name of the variable used for each product.

4GL	ESQL/C	ESQL/COBOL
SQLCA.SQLAWARN[4]	sqlca.sqlwarn.sqlwarn3	SQLWARN3 OF SQLWARN OF SQLCA

◆

Only the databases stored in your current directory, or in a directory specified in your **DBPATH** environment variable, are recognized. ◆

If you want to specify a database that does not reside in your current directory or in a directory specified by the **DBPATH** environment variable, you must follow the **DATABASE** keyword with a program or host variable that evaluates to the full pathname of the database (excluding the **.dbs** extension). ◆

I4GL

ESQL

ANSI

I4GL

ESQL

SE

SE

I4GL

ESQL

I4GL

The DATABASE statement can serve two purposes—one procedural and the other nonprocedural. The DATABASE statement makes the named database the current database (procedural), and tells the compiler where to find information about variables defined like (using the LIKE keyword) columns in a table (nonprocedural).

To serve the nonprocedural purpose, the DATABASE statement must occur outside any routine and precede the GLOBALS statements when you use indirect data typing with the LIKE clause. The *database name* must be expressed explicitly and not given as a program variable. You cannot use the EXCLUSIVE keyword in this context. If you use the DATABASE statement in this nonprocedural way, 4GL begins the main program block with *database name* as the current database.

If you close one database and open another in a program, you cannot define variables like columns in the second database.

If you do not have global variables defined like database columns, but still want to interact with a database, you can use the DATABASE statement in a purely procedural way. In this case, the DATABASE statement must occur within a function or the MAIN program block, and must follow any DEFINE statements within the function. Also in this case, *database name* can be a program variable, and you can use the EXCLUSIVE keyword. ♦

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.

I4GL

You cannot use the EXCLUSIVE keyword if you use the DATABASE statement outside a function or the MAIN program block. ♦

The following statement opens the **stores5** database on the **training** database server in exclusive mode:

```
DATABASE stores5@training EXCLUSIVE
```

If another user has the requested database open already, exclusive access is denied, an error is returned, and no database is opened.

References

In this manual, see the following statement: CLOSE DATABASE.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of database design and implementing the data model.

DEALLOCATE DESCRIPTOR

Purpose

Use the DEALLOCATE DESCRIPTOR statement to free a system descriptor area that was previously allocated for a specified *descriptor* or *descriptor variable*.

Syntax

```
ESQL — DEALLOCATE DESCRIPTOR — " — descriptor — " —
                                     |
                                     | descriptor
                                     | variable
```

descriptor is a string that identifies the system descriptor area that was allocated with the ALLOCATE DESCRIPTOR statement.

descriptor variable is an embedded variable name that identifies the system descriptor area that was allocated with the ALLOCATE DESCRIPTOR statement.

Usage

The DEALLOCATE DESCRIPTOR statement frees all the memory associated with the system descriptor area identified by *descriptor* or *descriptor variable*. All the value descriptors (including memory for data value in the value descriptors) also are freed.

A *descriptor* or *descriptor variable* can be reused after it is deallocated. Deallocation takes place automatically at the end of the program.

Deallocating a nonexistent *descriptor* or *descriptor variable* results in an error.

You cannot use the DEALLOCATE DESCRIPTOR statement to deallocate an `sqllda` structure. You can use it only to free the memory allocated for a system descriptor area. ♦

Following are examples of the DEALLOCATE DESCRIPTOR statement for three programming languages. In each pair, the first example shows an embedded variable name and the second example shows a quoted string that identifies the allocated system descriptor area.

Figure 7-7

Sample DEALLOCATE DESCRIPTOR statements in IBM Informix ESQL/C

```
$deallocate descriptor $descname;  
$deallocate descriptor "desc1";
```

Figure 7-8

Sample DEALLOCATE DESCRIPTOR statements in IBM Informix ESQL/COBOL

```
EXEC SQL DEALLOCATE DESCRIPTOR :DESCNAME END-EXEC  
EXEC SQL DEALLOCATE DESCRIPTOR "DESC1" END-EXEC
```

References

In this manual, see the following statements: ALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR.

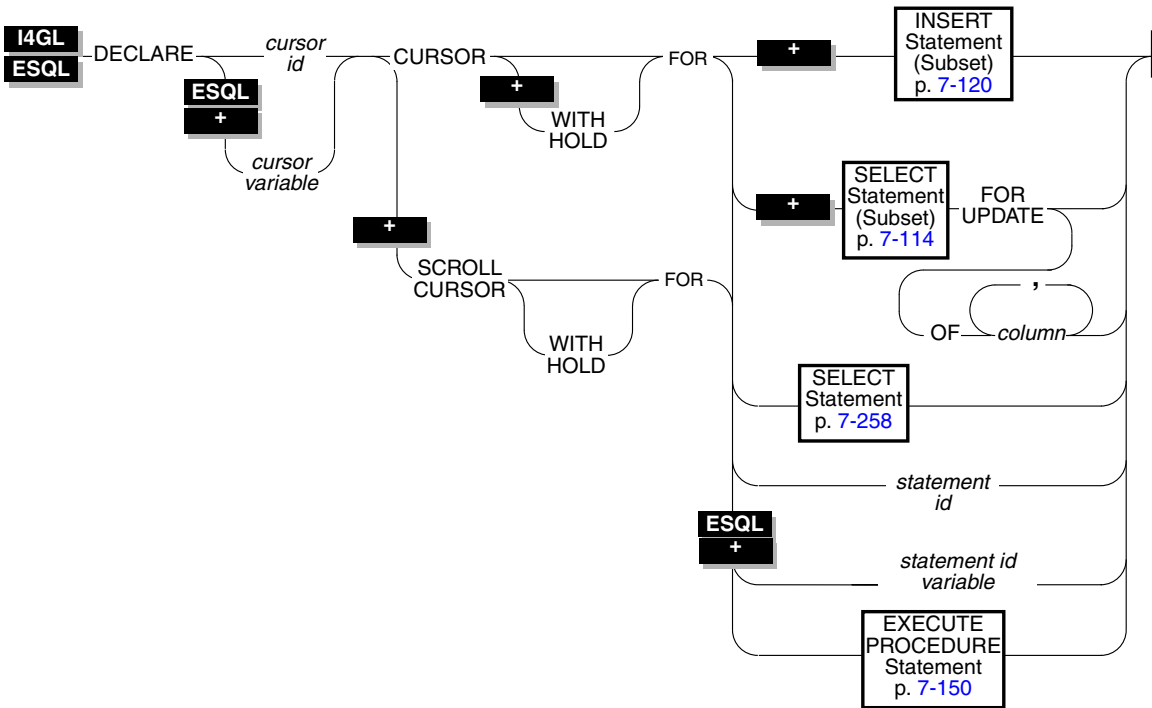
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of dynamic SQL.

DECLARE

Purpose

Use the DECLARE statement to define a cursor that represents the active set of rows specified by a SELECT or INSERT statement.

Syntax



column is a column that you can update through the cursor.

cursor id is the identifier of the cursor in other statements. The *cursor id* must conform to the same rules as any identifier, as described in the Identifier segment on page 7-399.

- cursor variable* is an embedded variable name that identifies the cursor in other statements. The *cursor variable* must conform to the same rules as any identifier, as described in the Identifier segment on page 7-399.
- statement id* is the identifier for a data structure that represents a prepared statement (see the PREPARE statement on page 7-218).
- statement id variable* is an embedded variable name that identifies a data structure that represents a prepared statement (see the PREPARE statement on page 7-218).

Usage

The DECLARE statement associates the cursor with a SELECT, INSERT, or EXECUTE PROCEDURE statement, or with the statement identifier (*statement id* or *id variable*) 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.

Used with a SELECT statement, the cursor 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. Used with an INSERT statement, the cursor represents the rows that the INSERT statement is to add to the database.

The DECLARE statement must precede any other statement that refers to the cursor during the execution of the program

The sum of the number of open cursors and the number of prepared statements that you can have at one time, in one process, is limited by the amount of free memory available in the system. Use `FREE statement id` or `FREE statement id variable` to release the resources held by a prepared statement; use `FREE cursor id` or `FREE cursor variable` to release resources held by a cursor.

A program can consist of one or more source code files. By default, the scope of a cursor is global to a program. This means that a cursor declared in one file can be referenced from another file.

E/C

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 of the files with the **-local** command line option. See your ESQL product manual for more information, restrictions, and performance issues when preprocessing with the **-local** option.

A variable used in place of the cursor name or statement identifier must be of the CHARACTER data type. In C, it must be defined as `$char`. ♦

E/CO

A variable used in place of the cursor name or statement identifier must be of the CHARACTER data type. In COBOL, such variables must be declared as a standard CHARACTER type. ♦

ESQL

You can declare multiple cursors using a single statement identifier. For example, the following IBM Informix ESQL/C example does not return an error.

Figure 7-9*Declaring multiple cursors in IBM Informix ESQL/C*

```
$PREPARE pid FROM "SELECT * FROM customer";
$DECLARE x CURSOR FOR pid;
$DECLARE y SCROLL CURSOR FOR pid;
$DECLARE z CURSOR WITH HOLD FOR pid;
```

If you include the **-ansi** compilation flag (or **DBANSIWARN** is set), warnings are generated for statements that use dynamic cursor names or dynamic statement id names. Some error checking performed in the compile phase prior to Version 5.0 is now done at run time. The typical checks are as follows:

- Illegal use of cursors (that is, normal cursors used as scroll cursors)
- Use of undeclared cursors
- Bad cursor or statement names (empty)

Checks for multiple declarations of a cursor of the same name are performed at compile time only if the cursor or statement is an identifier. For example, code such as the first example that follows results in a compile error, whereas the code in the second example does not.

Figure 7-10*Multiple cursor declaration that results in a compile error*

```
$DECLARE x CURSOR FOR SELECT * FROM customer;
. . .
$DECLARE x CURSOR FOR SELECT * FROM customer; -- error
```

Figure 7-11

Multiple cursor declaration that does not result in a compile error

```
$DECLARE x CURSOR FOR SELECT * FROM customer;  
. . .  
strcpy(s, "x");  
$DECLARE $s CURSOR FOR SELECT * FROM customer;  
◆
```

Overview of Cursor Types

Functionally, a cursor can be associated with a SELECT statement (a *select cursor*) or an INSERT statement (an *insert cursor*). You can use a select cursor to update or delete rows; then it is called an *update cursor*.

A cursor also can be associated with a statement identifier, enabling you to use a cursor with 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 (see the OPEN statement on page [7-207](#)).

Select Cursor

A select cursor enables you to scan multiple rows of data, moving data row by row into a set of receiving variables, as follows:

1. Use a DECLARE statement to define a cursor for the SELECT statement.
2. Open the cursor with the OPEN statement. The database server processes the query to the point of locating or constructing the first row of the active set.
3. Retrieve successive rows of data with the FETCH statement.
4. Close the cursor with the CLOSE statement when the active set is no longer needed.

Update Cursor

An update cursor is declared using the FOR UPDATE keywords. Using the update cursor, you can modify (update or delete) the current row.

In an ANSI-compliant database, you can update or delete data using a select cursor, as long as it follows the restrictions described on page 7-115. You do not need to use the FOR UPDATE keywords when you declare the cursor. ♦

Insert Cursor

An insert cursor increases processing efficiency (compared to 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 program and database server and 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). These structural characteristics are explained in the sections that follow.

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 only can read through the active set once each time it is opened. If you are using a sequential cursor, on each FETCH, the database server returns the contents of the current row and locates the next row in the active set.

The following IBM Informix ESQL/C example creates a sequential cursor.

Figure 7-12

Creating a sequential cursor in an IBM Informix ESQL/C program

```
$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. The database server implements a scroll cursor by creating 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, and fetch rows repeatedly without having to close and reopen the cursor. These abilities are discussed under the FETCH statement (see page 7-153).

The database server retains the active set for a scroll cursor 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 (available only with IBM Informix OnLine) or by locking the entire table in share mode during the transaction. (See the SET ISOLATION statement on page 7-307 and the LOCK TABLE statement on page 7-204.)

You cannot associate a scroll cursor with an INSERT statement and you cannot declare a scroll cursor with the FOR UPDATE keywords.

The following example creates a scroll cursor.

```
DECLARE sc_cur SCROLL CURSOR FOR
  SELECT * FROM orders
```

Figure 7-13
Creating a scroll cursor

Hold Cursor

If you use the WITH HOLD keywords, you create a hold cursor. A hold cursor remains open past the end of a transaction. You can declare both sequential and scroll cursors with the WITH HOLD keywords. The following example creates a hold cursor.

```
DECLARE hld_cur CURSOR WITH HOLD FOR
  SELECT customer_num, lname, city FROM customer
```

Figure 7-14
Creating a hold cursor

A hold cursor allows uninterrupted access to a set of rows across multiple transactions. Ordinarily, all cursors are closed at the end of a transaction; a hold cursor is not closed. You can use a hold cursor as shown in the following fragment of 4GL code. The 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 are scanned by the master cursor are the basis for updating the records pointed to by the detail cursor. In this example, 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.

Figure 7-15
Using a hold cursor in an IBM Informix 4GL program

```

DEFINE p_custnum INTEGER, p_orddate DATE, save_status INTEGER

PREPARE st_1 FROM
  "SELECT order_date ",
  "FROM orders WHERE customer_num = ? FOR UPDATE"
DECLARE c_detail CURSOR FOR st_1

DECLARE c_master CURSOR WITH HOLD FOR
  SELECT customer_num
  FROM customer WHERE city = "Pittsburgh"

OPEN c_master
IF status = 0 { the open worked } THEN
  FETCH c_master INTO p_custnum { discover first customer }
END IF
WHILE status = 0 { while no errors and not end of pittsburgh customers }
  BEGIN WORK { start transaction for customer p_custnum }
  OPEN c_detail USING p_custnum
  IF status = 0 { detail open succeeded } THEN
    FETCH c_detail INTO p_orddate { get first order }
  END IF
  WHILE status = 0 { while no errors and not end of orders }
    UPDATE orders SET order_date = "08/15/90"
    WHERE CURRENT OF c_detail
    IF status = 0 { update was ok } THEN
      FETCH c_detail INTO p_orddate { next order }
    END IF
  END WHILE
  IF status = NOTFOUND { correctly updated all found orders } THEN
    COMMIT WORK { make updates permanent, set status }
  ELSE { some failure in an update }
    LET save_status = status { save error for loop control }
    ROLLBACK WORK
    LET status = save_status { force loop to end }
  END IF

```

```
IF status = 0 { all updates, and the commit, worked ok } THEN
    FETCH c_master INTO p_custnum { next customer? }
END IF
END WHILE
CLOSE c_master
```

To close a hold cursor, use either the CLOSE statement to close the cursor explicitly or the CLOSE DATABASE statement to close it implicitly. (CLOSE DATABASE closes all cursors.)

Declaring an Update Cursor

The FOR UPDATE keywords notify the database server that updating is possible, causing it to use more stringent locking than with a select cursor. You are not allowed to modify data through a cursor without this clause. 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 allows you to perform updates that are not possible with the UPDATE statement, 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 being updated.

ANSI

All simple select cursors are potentially update cursors even if they are declared without the FOR UPDATE keywords. (See the restrictions on SELECT statements in the section that follows.) ♦

Subset of the SELECT Statement Associated with an Update Cursor

Not all SELECT statements can be associated with an update cursor. The SELECT statement included in the DECLARE statement (either directly or as a prepared statement) must conform to the following restrictions:

- You can select data from only one table.
- The statement cannot include any aggregate functions (AVG, COUNT, MAX, MIN, or SUM).
- The statement cannot include any of the following clauses or keywords:

DISTINCT	INTO TEMP	UNION
GROUP BY	ORDER BY	UNIQUE

For a complete description of SELECT syntax and usage, see the SELECT statement on page [7-258](#).

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 are fetched. Other programs can read the locked row, but no other program can place a promotable or write lock. Before the row is modified, the row lock is promoted to an exclusive lock.

SE

The IBM Informix SE database server does not use promotable locks. Before the program modifies a row, the database server obtains an exclusive lock on the row. ♦

Although it is possible to declare an update cursor WITH HOLD, the only reason for doing so is to break a long series of updates into smaller transactions. If an operation involves fetching and updating a very large number of rows, the lock table maintained by the database server can overflow. The usual way to prevent this overflow is to lock the entire table being updated. If that is not possible, 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, since COMMIT WORK releases all locks, even ones 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 modifies 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 two criteria:

- The SELECT statement can be processed using an index
- The columns listed are not part of the index used to process the SELECT statement

If the columns you intend to update are part of the index 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 even though it might be unnecessary.

The following example contains an 4GL DECLARE statement that restricts the columns for update.

Figure 7-16

Using the OF keyword in an IBM Informix 4GL program

```
DECLARE up_curs CURSOR FOR
  SELECT * FROM customer WHERE customer_num > 110
  FOR UPDATE OF fname, lname
```

The next example contains IBM Informix SQL/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 subsequent DELETE or UPDATE statement.

Figure 7-17

Using an update cursor in an IBM Informix ESQL/C program to delete rows

```

$declare q_curs cursor for
  select * from customer where lname matches $last_name
  for update;

$open q_curs;
for (;;)
{
  $fetch q_curs into $cust_rec;
  if (sqlca.sqlcode != 0)
    break;

  /* Display customer values and prompt for answer */

  if (answer[0] == 'y')
    $delete from customer where current of q_curs;
  if (sqlca.sqlcode != 0)
    break;
}
$close q_curs;

```

Associating a Cursor with a Prepared Statement

The PREPARE statement allows you to assemble the text of an SQL statement at run time and pass the statement text to the database server for execution. If you anticipate that a dynamically prepared SELECT statement could produce more than one row of data, the prepared statement must be associated with a cursor. (See the PREPARE statement on page 7-218 for more information about preparing SQL statements.)

The result of a PREPARE statement is a statement identifier (*statement id* or *id variable*) that is a data structure representing the prepared statement text. You declare a cursor for the statement text by associating a cursor with the statement identifier.

You can associate a sequential cursor with any prepared SELECT statement. You cannot associate a scroll cursor with a prepared INSERT statement or with a SELECT statement that was prepared to include a FOR UPDATE clause.

After a cursor is opened, used, and closed, a different statement can be prepared under the same statement identifier. In this way, it is possible to use a single cursor with different statements at different times.

The following example contains 4GL 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; then, the cursor `c_detail` is declared for `st_1`.

Figure 7-18

Declaring a cursor for a prepared statement in an IBM Informix 4GL program

```
PREPARE st_1 FROM
    "SELECT order_date ",
    "FROM orders WHERE customer_num = ?"
DECLARE c_detail CURSOR FOR st_1
```

If you want to modify data using a prepared SELECT statement, add a FOR UPDATE clause to the statement text you wish to prepare, as shown in the following 4GL example.

Figure 7-19

Declaring a cursor for a SELECT with a FOR UPDATE clause in an IBM Informix 4GL program

```
PREPARE sel_1 FROM "SELECT * FROM customer FOR UPDATE"
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 only begins when the BEGIN WORK statement is executed.

ANSI

In ANSI-compliant databases, transactions are always in effect. ♦

The database server enforces the following guidelines for 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 permits you to open and close a hold cursor for update outside a transaction; however, you should fetch all rows that pertain to a given modification and then perform the modification all within a single transaction. You cannot open and close cursors that are not hold or update cursors outside a transaction.

The following code example produces an error when the database server tries to execute the update line:

```
$declare q_curs cursor for
  select * from customer where lname matches $last_name
  for update;
$open q_curs;
$fetch q_curs into $cust_rec;
$begin work;
$update customer where current of q_curs;
$commit work;
```

The following code example does not produce an error when the database server tries to execute the update line:

```
$declare q_curs cursor for
  select * from customer where lname matches $last_name
  for update;
$open q_curs;
$begin work;
$fetch q_curs into $cust_rec;
$update customer where current of q_curs;
$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 declared for insert is an insert cursor. In a database that uses transactions, you cannot open an insert cursor outside a transaction unless it also was declared with hold.

Subset of INSERT Associated with a Sequential Cursor

You create an insert cursor by associating 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 4GL code that declares an insert cursor.

Figure 7-20

Declaring an insert cursor in an IBM Informix 4GL program

```
DECLARE ins_cur CURSOR FOR
  INSERT INTO customer VALUES (p_customer.*)
```

The next example contains IBM Informix ESQL/C code that declares an insert cursor.

Figure 7-21

Declaring an insert cursor in an IBM Informix ESQL/C program

```
$DECLARE ins_cur CURSOR FOR INSERT INTO stock VALUES
  ($stock_no, $manu_code, $descr, $u_price, $unit, $u_desc);
```

The insert cursor only inserts rows of data; it cannot be used for fetching 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. Data can be lost if the cursor is not closed properly.

Using an Insert Cursor with Hold

If you associate a hold cursor with an INSERT statement, you can break a long series of PUT statements into smaller sets of PUT statements by using transactions. 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

In this manual, see the following statements: CLOSE, DELETE, FETCH, FREE, INSERT, OPEN, PREPARE, PUT, SELECT, and UPDATE.

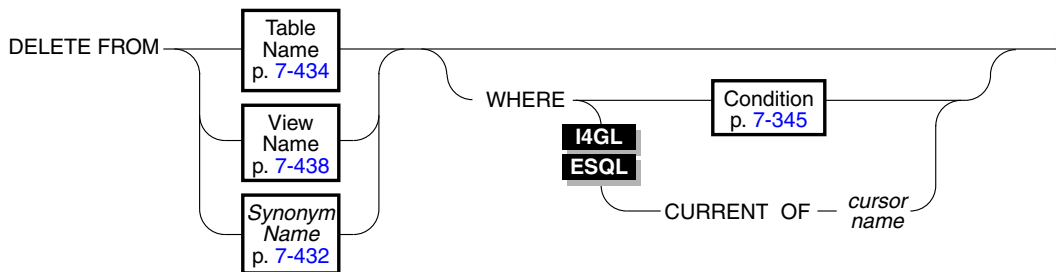
In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of cursors and data modification.

DELETE

Purpose

Use the DELETE statement to delete one or more rows from a table.

Syntax



cursor name is the name of the cursor that you have previously declared and positioned.

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.

Each row affected by a DELETE statement within a transaction is locked for the duration of the transaction; therefore, a single DELETE statement that affects a large number of rows locks the rows until the entire operation is complete. If the number of rows affected is very large, you might exceed the limits your operating system places on the maximum number of simultaneous locks. If this occurs, you can either reduce the scope of the DELETE statement or lock the entire table before you execute the statement.

DB

ISQL

ANSI

If you specify a view name, the view must be updatable. See [“Updating through Views” on page 7-100](#) for an explanation of an updatable view.

If you omit the WHERE clause while working within the SQL Menu, IBM Informix SQL 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. ♦

Statements are always within an implicit transaction in an ANSI-compliant database; therefore, you cannot have a DELETE statement outside a transaction. ♦

WHERE Clause

Use the WHERE clause to specify one or more rows that you want deleted. The WHERE conditions are the same as those 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

ISQL

If you include a WHERE clause that selects all rows in the table, IBM Informix SQL gives no prompt and deletes all rows. ♦

I4GL

ESQL

CURRENT OF Clause

To use the CURRENT OF clause, you must previously have 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, there is no current row; you cannot use the cursor to delete or update a row until you reposition the cursor with a FETCH statement. ♦

ANSI

I4GL

ESQL

All select cursors are potentially update cursors in ANSI-compliant databases. You can use the CURRENT OF clause with any select cursor. ♦

References

In this manual, see the following statements: INSERT, UPDATE, DECLARE, and FETCH.

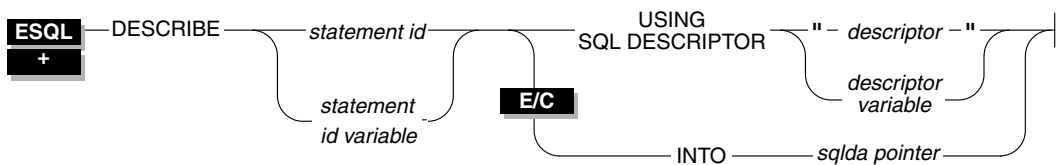
In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of cursors and data modification.

DESCRIBE

Purpose

Use the DESCRIBE statement to obtain information about a prepared statement before you execute it. The DESCRIBE statement returns the prepared statement type and, for a SELECT or INSERT statement, the number, data types, and size of the values, and the name of the column or expression, returned by the query. The information can be stored in a system descriptor area or, in ESQL/C, in an **sqlda** structure.

Syntax



- descriptor* is a quoted string that identifies an allocated system descriptor area.
- descriptor variable* is an embedded variable name that identifies an allocated system descriptor area.
- sqlda pointer* points to an **sqlda** structure.
- statement id* is the identifier for a data structure that represents a prepared statement. (See the PREPARE statement on page 7-218.)
- statement id variable* is an embedded variable name that identifies a data structure that represents a prepared statement. (See the PREPARE statement on page 7-218.)

Usage

The DESCRIBE statement allows you to determine at run time the type of statement that has been prepared and the number and types of data that a prepared query will return when 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 as input a statement identifier from a PREPARE statement. When the DESCRIBE statement executes, the database server sets the value of the SQLCODE field of the SQLCA (see [Chapter 5, “Error Handling with 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 is the most common situation. 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 IBM Informix ESQL/C, the constant names are defined in the `sqlstype.h` header file. A printed list of the possible values and their constant names appears in the manual for each embedded-language product.

The DESCRIBE statement uses the SQLCODE field differently than any other statement, possibly returning a nonzero value when it executes successfully. You can revise standard error checking routines to accommodate this, if desired.

Checking for Existence of a WHERE Clause

If the DESCRIBE statement detects that a prepared statement contains an UPDATE or DELETE statement without a WHERE clause, the DESCRIBE statement sets the following SQLCA variable to W.

ESQL/C **sqlca.sqlwarn.sqlwarn4**

ESQL/COBOL **SQLWARN4 OF SQLWARN OF SQLCA**

Without a WHERE clause, the update or delete action is applied to the entire table. By checking this variable, you can avoid unintended global changes to your table.

Describing Values Returned by SELECT or Required for INSERT

If the prepared statement text includes a SELECT statement without an INTO TEMP clause or an INSERT statement, the DESCRIBE statement also returns a description of each column or expression included in the SELECT or INSERT list. These descriptions are stored in a pointer to an **sqllda** structure or in a system descriptor area.

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

See [Chapter 6, “Using Descriptors,”](#) for more information on the **sqllda** structure and descriptors.

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 some other, compatible type. This change causes data conversion to take place when the data is fetched.

In addition to [Chapter 6](#) of this manual, see the manual for your embedded-language product for further information about interpreting and using the data contained in the `sqllda` data structure and the system descriptor area.

USING SQL DESCRIPTOR Clause

The USING SQL DESCRIPTOR clause allows you to store the description of a SELECT or INSERT list in a system descriptor area created by 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 an `sqllda` structure.

The DESCRIBE statement sets the COUNT field in the system descriptor area to the number of values in the SELECT or INSERT list. If COUNT is greater than the number of item descriptors (*occurrences*) 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.

INTO sqllda pointer Clause

The INTO *sqllda pointer* clause allows you to store the description of a SELECT or INSERT list in an `sqllda` pointer. The DESCRIBE statement sets the `sqllda.sqlld` variable to the number of values in the SELECT or INSERT list. The `sqllda` structure also contains an array of data descriptors (`sqllvar` structures), one for each value in the SELECT or INSERT list. After a DESCRIBE statement is executed, the `sqllda.sqllvar` structure has the TYPE, LENGTH, and NAME fields set.

The DESCRIBE statement allocates memory for an `sqllda` pointer once it is declared in a program. However, the application program must designate the storage area of the `sqllvar.sqlldata` fields. ♦

E/C

These products do not support pointers to an `sqllda` structure; they return an error if you try to execute a DESCRIBE statement that uses one. Only system descriptor areas that are allocated with the ALLOCATE DESCRIPTOR statement can be used in a DESCRIBE statement in IBM Informix ESQL/COBOL. You can view the contents of the columns by executing a GET DESCRIPTOR statement following a DESCRIBE statement on the specified system descriptor. ♦

The following pairs of examples show the use of a system descriptor in a DESCRIBE statement in three IBM Informix ESQL products. In the first example in each pair, the descriptor is a quoted string; in the second example in each pair it is an embedded variable name.

Figure 7-22

Sample DESCRIBE operation with system descriptor statements in ESQL/C

```
main()
{
. . .
$ ALLOCATE DESCRIPTOR "desc1" WITH MAX 3;
$ PREPARE curs1 FROM "SELECT * FROM tab";
$ DESCRIBE curs1 USING SQL DESCRIPTOR "desc1";
}

$ DESCRIBE curs1 USING SQL DESCRIPTOR $desc1var;
```

Figure 7-23

Sample DESCRIBE operation with system descriptor statements in ESQL/COBOL

```
EXEC SQL ALLOCATE DESCRIPTOR "DESC1" WITH MAX 3 END-EXEC.
EXEC SQL PREPARE CURS1 FROM "SELECT * FROM TAB" END-EXEC.
EXEC SQL DESCRIBE CURS1 USING SQL DESCRIPTOR "DESC1" END-EXEC.

EXEC SQL DESCRIBE CURS1 USING SQL DESCRIPTOR :DESC1VAR END-EXEC.
```

References

In this manual, see the following statements for further information about using dynamic management statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of the DESCRIBE statement.

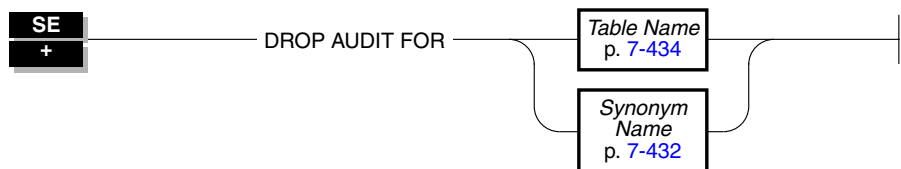
For further information about how to use an **sqlda** pointer or a system descriptor area if you intend to use a `FETCH...USING DESCRIPTOR` or an `INSERT...USING DESCRIPTOR` statement, refer to the manual for your application development tool. Also see [Chapter 6](#) for information on the **sqlda** structure.

DROP AUDIT

Purpose

Use the DROP AUDIT statement to delete an audit trail file.

Syntax



Usage

When you finish making a backup of your database files, use the DROP AUDIT statement to remove the old audit trail file. Use the CREATE AUDIT statement to start a new audit trail for a table.

You must own the table or have DBA status to use the DROP AUDIT statement.

The following example assumes that you have just backed up the **stores5** database. It removes the existing audit trail on the **orders** table.

```
DROP AUDIT FOR orders
```

References

In this manual, see the following statements: CREATE AUDIT and RECOVER TABLE.

In the *IBM Informix SE Administrator's Guide*, see the discussion on audit trails.

DROP DATABASE

Purpose

Use the DROP DATABASE statement to delete an entire database, including all system catalogs, indexes, and data.

Syntax



Usage

You must have DBA status 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 users of the database must first execute the CLOSE DATABASE statement.

The DROP DATABASE statement cannot appear in a multistatement PREPARE statement.

The following statement drops the **stores5** database:

```
DROP DATABASE stores5
```

When you drop a database with transactions, the transaction log file associated with the database is removed. ♦

Use this statement with caution. If you have DBA privilege, DB-Access and IBM Informix SQL do not prompt you to verify that you want to delete the entire database. ♦

SE

DB

ISQL

I4GL

ESQL

SE

SE

I4GL

ESQL

You can use a simple database name in a program or host variable, or you can use the full database server and database name. See the explanation of Database Name on page 7-362 for more information. ♦

The DROP DATABASE statement does not remove the database directory if it includes any files other than those created for database tables and their indexes.

You can specify the full pathname of the database in quotes, as shown in the following example:

```
DROP DATABASE "/u/training/stores5"
```

You cannot use a ROLLBACK WORK statement to undo a DROP DATABASE statement. If you roll back a transaction that contains a DROP DATABASE statement, the database is not re-created and you do not receive an error message. ♦

You can specify a database that is not in your local directory or DBPATH by putting the full operating system file in a variable for the database name.

```
LET db_var = "/u/training/stores5"
DROP DATABASE db_var
```

♦

References

In this manual, see the following statements: CREATE DATABASE and CLOSE DATABASE.

DROP INDEX

Use the DROP INDEX statement to remove an index.

Syntax



Usage

You must be the owner of the index or have DBA status to use the DROP INDEX statement.

The following example drops the index **o_num_ix** owned by **joed**. The **stores5** database must be the current database.

```
DROP INDEX stores5:joed.o_num_ix
```

You cannot use the DROP INDEX statement on a column or columns to drop a unique constraint created with a CREATE TABLE statement; you must use the ALTER TABLE statement to remove indexes 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 using the following format:

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

where *tabid* and *constraint_id* are from the **sysstables** and **sysconstraints** system catalog tables, respectively. The **idxname** (index name) column in **sysconstraints** is then updated to reflect this change.

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. ♦

You cannot use a ROLLBACK WORK statement to undo a DROP INDEX statement. If you roll back a transaction that contains a DROP INDEX statement, the index is not re-created and you do not receive an error message. ♦

DB

ESQL

SE

References

In this manual, see the following statements: ALTER TABLE, CREATE INDEX, and CREATE TABLE.

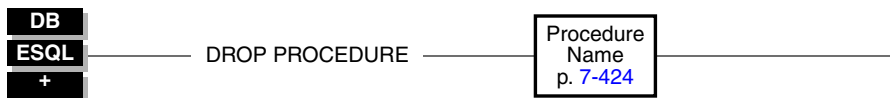
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of indexes.

DROP PROCEDURE

Purpose

Use the DROP PROCEDURE statement to remove a procedure from the database.

Syntax



Usage

You must be the owner of the procedure or have DBA status to use the DROP PROCEDURE statement.

Dropping the procedure removes the text and executable versions of the procedure.

You cannot use a ROLLBACK WORK statement to undo a DROP PROCEDURE statement. If you roll back a transaction that contains a DROP PROCEDURE statement, the procedure is not re-created and you do not receive an error message. ♦

References

In this manual, see the CREATE PROCEDURE statement.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of using procedures.

SE

DROP SYNONYM

Purpose

Use the DROP SYNONYM statement to remove a previously defined synonym.

Syntax



Usage

You must be the owner of the synonym or have DBA status to use the DROP SYNONYM statement.

The following statement drops the synonym **nj_cust**, owned by **cathyg**:

```
DROP SYNONYM cathyg.nj_cust
```

If a table is dropped, any synonyms in the same database as the table that refer to the table also are 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, giving the new object the name of the dropped table. The old synonym then refers to the new object. See the CREATE SYNONYM statement for a complete discussion of synonym chaining.

SE

You cannot use a ROLLBACK WORK statement to undo a DROP SYNONYM statement. If you roll back a transaction that contains a DROP SYNONYM statement, the synonym is not re-created and you do not receive an error message. ♦

Reference

In this manual, see the following statement: `CREATE SYNONYM`.

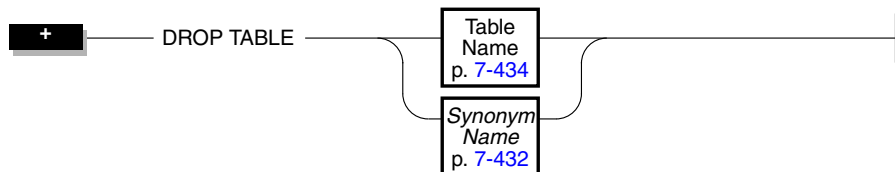
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of synonyms.

DROP TABLE

Purpose

Use the DROP TABLE statement to remove a table, along with its associated indexes and data.

Syntax



Usage

You must be the owner of the table or have DBA status to use the DROP TABLE statement.

Use the DROP TABLE statement with caution. When you remove a table, you also delete the data stored in it, the indexes or constraints on the columns (including all of the referential constraints placed on its columns), any local synonyms assigned to it, and any authorizations you have granted on the table. You also drop all views based on the table. You do not remove any synonyms for the table that have been created in an external database.

You cannot drop any of the system catalog tables. You cannot drop a table that is not in the current database.

If you issue a DROP TABLE statement in DB-Access or IBM Informix SQL, you are not prompted to verify that you want to delete an entire table. ♦

ISQL

DB

You cannot use a ROLLBACK WORK statement to undo a DROP TABLE statement. If you roll back a transaction that contains a DROP TABLE statement, the table is not re-created and you do not receive an error message. ♦

The following example deletes two tables. Both tables are within the current database and owned by the current user.

```
DROP TABLE customer
DROP TABLE stores5@acctng:joed.state
```

Reference

In this manual, see the following statements: CREATE TABLE, DROP DATABASE.

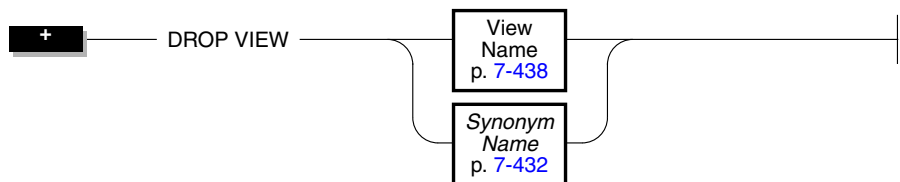
In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of data integrity and creating a table.

DROP VIEW

Purpose

Use the DROP VIEW statement to remove a view from the database.

Syntax



Usage

You must own the view or have DBA status to use the DROP VIEW statement.

When you drop *view name*, you also drop all views that have been defined in terms of *view name*. You can determine which, if any, views depend on another view by querying the **sysdepend** system catalog table.

The following statement drops the view named **cust1**:

```
DROP VIEW cust1
```

SE

You cannot use a ROLLBACK WORK statement to undo a DROP VIEW statement. If you roll back a transaction that contains a DROP VIEW statement, the view is not re-created and you do not receive an error message. ♦

References

In this manual, see the following statements: CREATE VIEW and DROP TABLE.

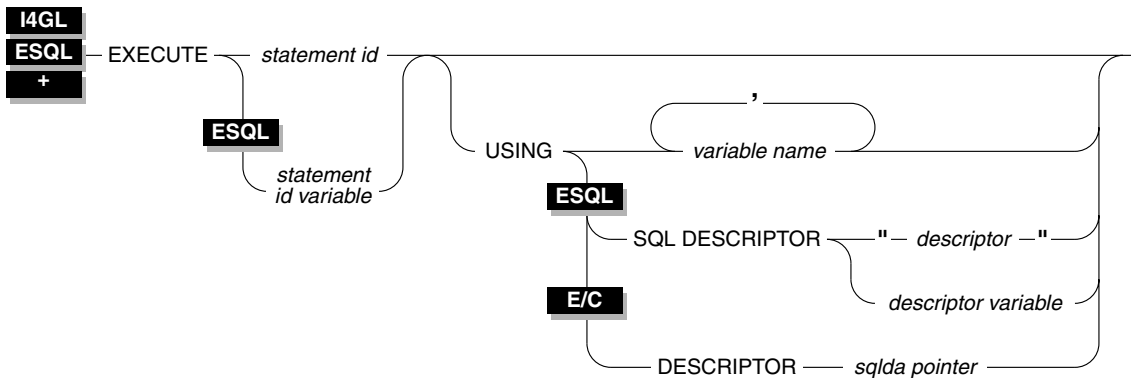
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of views.

EXECUTE

Purpose

Use the EXECUTE statement to run a previously prepared statement.

Syntax



descriptor is a quoted string that identifies the system descriptor area that was previously allocated.

descriptor variable is an embedded variable name that identifies the system descriptor area that was previously allocated.

sqlda pointer is an IBM Informix ESQL/C pointer to an **sqlda** structure that describes the undefined values in the prepared statement.

statement id is an SQL statement identifier defined in a previous PREPARE statement in the same module.

- statement id variable* is an embedded variable name that identifies the SQL statement defined in a previous PREPARE statement in the same module.
- variable name* is an IBM Informix 4GL program variable or an IBM Informix ESQL host variable to be substituted as a value for a question mark (?) placeholder required by the prepared statement.

Usage

The EXECUTE statement passes a prepared SQL statement to the database server for execution. If the statement contained ? 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 except a (prepared) SELECT statement. A prepared SELECT statement returns rows of data; you should use the DECLARE, OPEN, and FETCH cursor statements to retrieve the data rows. (You can, however, use EXECUTE on a prepared SELECT INTO TEMP statement.)

Following an EXECUTE statement, the SQLCA (see [Chapter 5, “Error Handling with SQLCA,”](#) for information about the SQLCA) might reflect an error in the EXECUTE statement (for example, error -260, Cannot execute a SELECT statement that is PREPARED - must use cursor), but usually it reflects the success or failure of the executed statement itself.

An example of an EXECUTE statement within an IBM Informix 4GL program follows.

Figure 7-24

Using an EXECUTE statement in an IBM Informix 4GL program

```
PREPARE sel_1 FROM
  "DELETE FROM customer ",
  "WHERE customer_num = 119"
EXECUTE sel_1
```

A program can consist of one or more source code files. By default, the scope of a statement identifier is global to the program. This means that a statement identifier executed 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 should preprocess all the files with the **-local** command line option. See your ESQL product manual for more information, restrictions, and performance issues when preprocessing with the **-local** option.

USING Clause

The USING clause specifies values that are to replace ? placeholders in the prepared statement. Providing values in the EXECUTE statement that replace the ? placeholders in the prepared statement is sometimes called *parameterizing* the prepared statement.

You can specify any of the following items to replace the ? placeholders in a statement before you execute it:

- A host or program variable name (if the number and data type of the question marks are known at compile time)
- A system descriptor that identifies a system descriptor area ♦
- A descriptor that is a pointer to an **sqllda** structure ♦

ESQL

E/C

Supplying Parameters Through Host or Program Variables

You must supply one *variable name* for each placeholder. The data type of each variable must be compatible with the corresponding value required by the prepared statement.

The *variable name* can include an indicator variable, provided its use is appropriate at the corresponding point in the prepared statement. ♦

The following two examples execute the same prepared UPDATE statement as expressed in IBM Informix 4GL and IBM Informix ESQL/C.

Figure 7-25

Sample EXECUTE statement in IBM Informix 4GL

```
LET stm_1 = "UPDATE orders SET order_date = ? ",
           "WHERE po_num = ?"
PREPARE statement_1 FROM stm_1
EXECUTE statement_1 USING x_o_date, x_po_num
```

ESQL

Figure 7-26*Sample EXECUTE statement in IBM Informix ESQL/C*

```
stm1 = "UPDATE orders SET order_date = ? WHERE po_num = ?";
$PREPARE statement_1 from stm1;
$EXECUTE statement_1 USING $order_date:ord_ind, $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 described by the system descriptor area are used to replace ? placeholders in the PREPARE statement. This method is similar to using the USING keyword with a list of variables, except that your program has full control over the memory location of the data values.

The COUNT field corresponds to the number of dynamic parameters in the prepared statement. The value of COUNT must be less than or equal to the value of the occurrences specified when the system descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.

For more information on system descriptors, see [Chapter 6](#) in this manual and the manual for the IBM Informix ESQL product you are using.

The following examples show how to execute prepared statements using system descriptors in three IBM Informix ESQL products.

Figure 7-27*Sample EXECUTE USING SQL DESCRIPTOR statement in ESQL/C*

```
$ EXECUTE prep_stmt USING SQL DESCRIPTOR "desc1";
```

Figure 7-28*Sample EXECUTE USING SQL DESCRIPTOR statement in ESQL/COBOL*

```
EXEC SQL EXECUTE PREP_STMT USING SQL DESCRIPTOR "DESC1" END-EXEC
```

◆

Supplying Parameters Through an sqlda Structure

You can specify the **sqlda** pointer in the USING DESCRIPTOR clause of the EXECUTE statement. Each time that the EXECUTE statement is run, the values described by the **sqlda** structure are used to replace ? placeholders in the PREPARE statement. This method is similar to employing the USING keyword with a list of variables, except that your program has full control over the memory location of the data values.

For more information on the **sqlda** structure, see the manual for the version of IBM Informix ESQL/C you are using and [Chapter 6](#) in this manual.

The following example shows how to execute a prepared statement using an **sqlda** structure in IBM Informix ESQL/C.

Figure 7-29

Sample EXECUTE USING DESCRIPTOR statement in ESQL/C

```
$EXECUTE prep_stmt USING DESCRIPTOR pointer2;  
◆
```

References

In this manual, see the following statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, EXECUTE IMMEDIATE, GET DESCRIPTOR, PREPARE, PUT, and SET DESCRIPTOR.

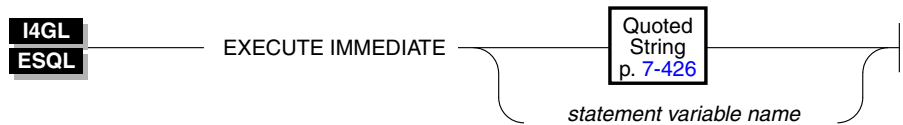
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of the EXECUTE statement.

EXECUTE IMMEDIATE

Purpose

Use the EXECUTE IMMEDIATE statement to perform the functions of the following SQL statements in one step: PREPARE, EXECUTE, and FREE.

Syntax



statement variable name is a program or host variable that contains a character string that consists of one or more SQL statements

Usage

The quoted string is a character string made up of one or more SQL statements. The string, or the contents of the *statement variable name*, is parsed and executed if correct; then, all data structures and memory resources are released immediately. In the usual method of dynamic execution, these functions are distributed among the following statements: PREPARE, EXECUTE, and FREE.

The EXECUTE IMMEDIATE statement makes it easy to dynamically execute a single, simple SQL statement that is constructed during program execution. For example, you could obtain the name of a database from program input, then construct the DATABASE statement as a program variable, and then use EXECUTE IMMEDIATE to execute the statement, opening the database.

Restricted Statement Types

You cannot use the EXECUTE IMMEDIATE statement to execute the following SQL statements:

CLOSE	EXECUTE	OPEN	SELECT
DECLARE	FETCH	PREPARE	WHENEVER

Use a PREPARE statement to execute a dynamically constructed SELECT statement.

The following restrictions apply to the statement contained in the quoted string or in *statement variable name*:

- The statement cannot contain a host-language comment.
- Names of host-language variables are not recognized as such in prepared text. The only identifiers that you can use are names defined in the database, such as table names and columns.
- The statement cannot reference a host variable list or a descriptor; hence it should not contain any ? placeholders, such as those allowed with a PREPARE statement.
- The text should not include any embedded SQL statement prefix or terminator, such as the dollar sign or semicolon, or the words EXEC SQL. ♦

In the following IBM Informix 4GL example, the user is prompted for the name of a table to drop. The statement text is formed using an IBM Informix 4GL character expression.

Figure 7-30

Sample EXECUTE IMMEDIATE statement in IBM Informix 4GL

```
DEFINE tabname CHAR(18)
PROMPT "Drop which table?" FOR tabname
EXECUTE IMMEDIATE "DROP TABLE ", tabname
```

An example of EXECUTE IMMEDIATE using IBM Informix ESQL/C follows.

Figure 7-31

Sample EXECUTE IMMEDIATE statement in IBM Informix ESQL/C

```
sprintf(cdb_text, "create database %s", usr_db_id);
$EXECUTE IMMEDIATE $cdb_text;
```

References

In this manual, see the following statements: EXECUTE, FREE, and PREPARE.

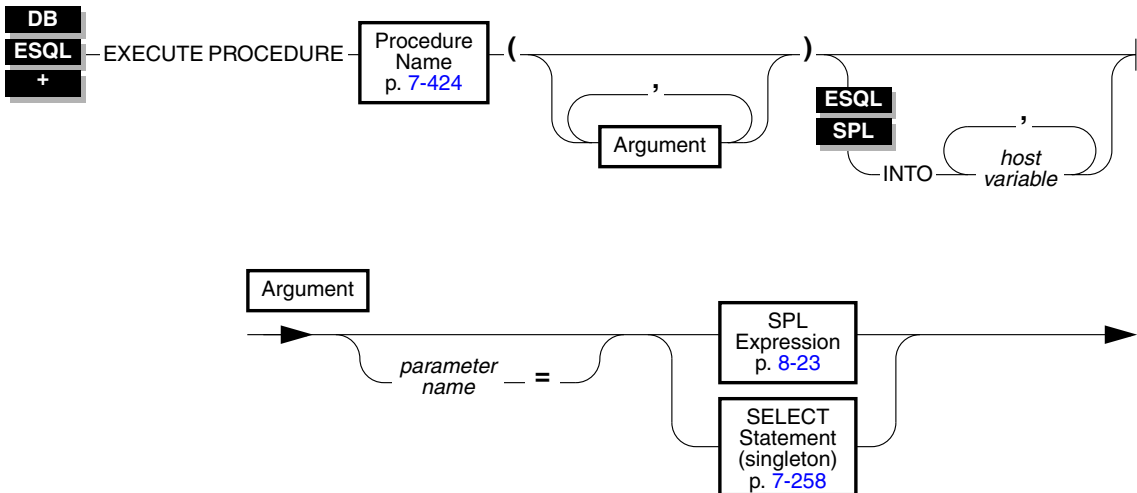
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of quick execution.

EXECUTE PROCEDURE

Purpose

Use the EXECUTE PROCEDURE statement to execute a procedure from the DB-Access interactive editor, an embedded-language program, or another stored procedure.

Syntax



host variable is a variable defined within the calling program.

parameter name is the name of the parameter as defined by its CREATE PROCEDURE statement.

Usage

The EXECUTE PROCEDURE statement invokes a procedure called Procedure Name.

If there are more arguments in an EXECUTE PROCEDURE statement than are expected by the called procedure, an error is returned.

If there are fewer arguments in an EXECUTE PROCEDURE statement than are expected by the called procedure, the arguments are said to be missing. Missing arguments are initialized to their corresponding default values, if default values were specified. (See CREATE PROCEDURE on page 7-58.) 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.

Procedure arguments are bound to procedure parameters by name or position, but not both. That is, you can use *parameter name* = syntax for none or all of the arguments specified in one EXECUTE PROCEDURE statement.

For example, both of the procedure calls are valid for a procedure that expects three character arguments: t, n, and d, as in the following example:

```
EXECUTE PROCEDURE add_col (t="customer", d="integer", n="newint")
EXECUTE PROCEDURE add_col("customer","newint","integer")
```

ESQL

If the EXECUTE PROCEDURE statement returns more than one row, it must be enclosed within an SPL FOREACH loop or accessed through a cursor. ♦

INTO Clause

The *host variable* 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 procedure, the list contains procedure variables. ♦

ESQL

SPL

References

In this manual, see the following statements: CREATE PROCEDURE, GRANT, and CALL.

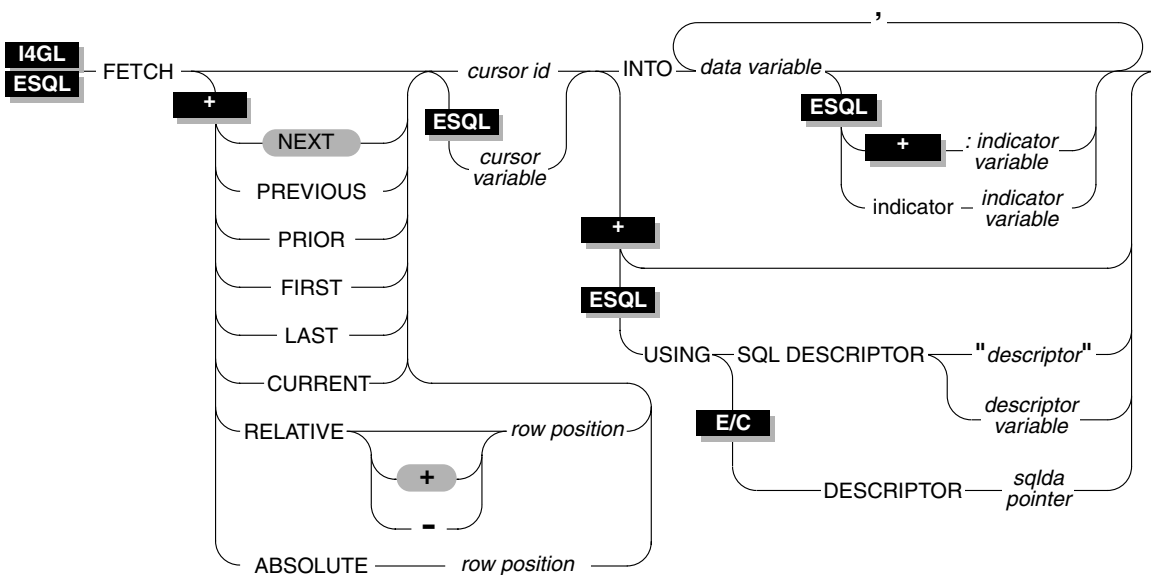
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of creating the data model.

FETCH

Purpose

Use the FETCH statement to move a cursor to a new row in the active set and to retrieve the row values into memory for use by the program.

Syntax



cursor id is the identifier of a cursor that was created in an earlier DECLARE statement.

cursor variable is an embedded variable name that identifies a cursor that was created in an earlier DECLARE statement.

data variable is a program variable or host object that receives one value from the fetched row.

<i>descriptor</i>	is a string that identifies the system descriptor area that was allocated with the ALLOCATE DESCRIPTOR statement.
<i>descriptor variable</i>	is an embedded variable name that identifies the system descriptor area that was allocated with the ALLOCATE DESCRIPTOR statement.
<i>indicator variable</i>	is a program variable that receives a return code if null data is placed in the corresponding <i>data variable</i> .
<i>row position</i>	is an integer or variable that contains an integer value, giving the position of the desired row in the active set of rows.
<i>sqlda pointer</i>	is a pointer to an sqlda structure that receives the values from the fetched row.

Usage

The FETCH statement is one of four statements used for queries that return more than one row from the database. The four statements, DECLARE, OPEN, FETCH, and CLOSE, are used in the following sequence:

- Declare a cursor to control the active set of rows.
- Open the cursor to begin execution of the query.
- Fetch from the cursor to retrieve the contents of each row.
- Close the cursor to break the association between the cursor and the active set.

A cursor is created as either a sequential cursor or scroll cursor. The way the database server creates and stores members of the active set and then fetches rows from the active set differs depending on whether the cursor is a sequential cursor or scroll cursor. (See the DECLARE statement on page [7-107](#) for details on the types of cursors.)

In X/Open mode, if a cursor direction value (such as NEXT or RELATIVE) is specified, a warning message appears indicating that the statement does not conform to X/Open standards. ♦

X/O

FETCH with a Sequential Cursor

A sequential cursor can fetch only the next row in sequence from the active set. The only keyword option available to a sequential cursor is the default value, NEXT. A sequential cursor only can read through a table once each time it is opened. The following example in IBM Informix ESQL/C illustrates the use of a sequential cursor:

```
$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.

Since the sequential cursor can retrieve only the next row, it is frequently possible for the database server to 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 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. You use keywords to specify a particular row that you want retrieved.

NEXT	Retrieves the next row in the active set.
PREVIOUS	Retrieves the previous row in the active set.
PRIOR	Synonymous with PREVIOUS; retrieves the previous row in the active set.
FIRST	Retrieves the first row in the active set.
LAST	Retrieves the last row in the active set.

CURRENT	Retrieves the current row in the active set (the same row as returned by the preceding FETCH statement from the scroll cursor).
RELATIVE	Retrieves the <i>n</i> th row relative to the current cursor position in the active set, where <i>n</i> is supplied by <i>row position</i> . A negative value indicates the <i>n</i> th row prior to the current cursor position. If <i>row position</i> is zero, the current row is fetched.
ABSOLUTE	Retrieves the <i>n</i> th row in the active set, where <i>n</i> is supplied by <i>row position</i> . Absolute row positions are numbered from 1.

The following 4GL examples illustrate some uses of the FETCH statement.

Figure 7-32

Sample FETCH statements in IBM Informix 4GL

```
FETCH PREVIOUS q_curs INTO orders.*  
  
FETCH LAST q_curs INTO orders.*  
  
FETCH RELATIVE -10 q_curs INTO orders.*  
  
PROMPT "Which row?" FOR row_num  
FETCH ABSOLUTE row_num q_curs INTO orders.*
```

Row Numbers

The row numbers used with the ABSOLUTE keyword are valid only while the cursor is open. Do not confuse them with rowid values. A rowid value is based on the position of a row in its table, and remains valid until the table is rebuilt. A row number for a FETCH statement is based on the position of the row in the active set of the cursor; the next time the cursor is opened, different rows may 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 is closed, because it cannot be sure which row the program will ask for next. When a scroll cursor is opened, the database server implements the active set as a temporary table, although it may 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 has fetched all the rows. Rows that are never fetched are usually not created or saved.

Specifying Where Values Go in Memory

Each value from the select list of the query must be returned into a memory location for the program to use. You can specify these destinations in one of the following ways:

- Using the INTO clause of a SELECT statement
- Using the INTO clause of a FETCH statement
- Using a system descriptor
- Using an `sqllda` structure ♦

E/C

Using the INTO Clause of SELECT

The SELECT statement that is associated with the cursor can contain an INTO clause that specifies which program variables are to receive the values. You only can use this method when the SELECT statement is written as part of the declaration of the cursor (see the DECLARE statement on page 7-107). In this case, the FETCH statement cannot contain an INTO clause. Here is an example in IBM Informix 4GL.

Figure 7-33

Using the INTO clause of SELECT to specify program variables in IBM Informix 4GL

```
DECLARE ord_date CURSOR FOR
  SELECT order_num, order_date, po_num
     INTO o_num, o_date, o_po
OPEN ord_date
FETCH NEXT ord_date
```

ESQL

You should use an indicator variable if there is the possibility that data returned from the SELECT is NULL. See your embedded-language product manual for more information about indicator variables. ♦

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 has the advantage that you can store different rows in different memory locations.

You cannot use an array of host variables in the INTO clause.

In the following IBM Informix 4GL example, a series of complete rows is fetched into a program array.

Figure 7-34

Fetching a series of rows with IBM Informix 4GL

```
DEFINE cust_list ARRAY[100] OF RECORD LIKE customer.*
DEFINE wanted_state LIKE customer.state
DEFINE row_count SMALLINT
DECLARE cust CURSOR FOR
    SELECT * FROM customer WHERE state = wanted_state
PROMPT "Enter 2-letter state code:" FOR wanted_state
OPEN cust
LET row_count = 0
WHILE status = 0
    LET row_count = row_count + 1
    FETCH NEXT cust INTO cust_list[row_count].*
END WHILE
CLOSE 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

ESQL

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. The values returned by the FETCH statement can then be transferred from the system descriptor area into host variables by using the GET DESCRIPTOR statement.

For more information, see [Chapter 6](#) in this manual, as well as the manual for the IBM Informix ESQL product you are using.

Following are examples of the use of system descriptors in three IBM Informix embedded-language products.

Figure 7-35

Sample `FETCH USING SQL DESCRIPTOR` statement in `ESQL/C`

```
$ FETCH selcurs USING SQL DESCRIPTOR "desc";
```

Figure 7-36

Sample `FETCH USING SQL DESCRIPTOR` statement in `ESQL/COBOL`

```
EXEC SQL FETCH SEL_CURS USING SQL DESCRIPTOR "DESC" END-EXEC.
```

Using an `sqlda` Structure

E/C

You can supply destinations using a pointer to an `sqlda` structure. This structure contains data descriptors, each one specifying the data type and memory location for one selected value. For more information, see [Chapter 6](#) in this manual, as well as the *IBM Informix ESQL/C Programmer's Manual*. The keywords `USING DESCRIPTOR` introduce the name of the `sqlda` pointer structure.

When you create a `SELECT` statement dynamically, you cannot use an `INTO host-variable` clause because you cannot name host variables in a prepared statement. If you are certain of the number and type of values in the select list, you can use an `INTO host-variable` clause in the `FETCH` statement. However, if the query was generated by user input, you might not be certain of the number and type of values being selected. In this case, you must use an `sqlda` pointer structure as follows:

- Use the `DESCRIBE` statement to fill in the `sqlda`.
- Allocate memory to hold the data values.
- Name the `sqlda` in the `FETCH` statement.

Figure 7-37

Sample `FETCH USING DESCRIPTOR` statement in `IBM Informix ESQL/C`

```
$ FETCH selcurs USING DESCRIPTOR pointer2;
```

◆

Fetching a Row for Update

The FETCH statement does not ordinarily lock a row that is fetched. Thus, the fetched row can be modified (updated or deleted) by another process 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 is closed or the current transaction ends. Other programs also can 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 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 will be unchanged if another user tries to modify it using the WHERE CURRENT OF clause of UPDATE or DELETE.

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 may or may not be released when you fetch another row, depending on the isolation level you have set. The lock on an unchanged row is released as soon as another row is fetched, unless you are using Repeatable Read isolation (see the SET ISOLATION statement on page 7-307).

Tip: You can hold locks on additional rows even when Repeatable Read isolation is not in use or 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 should 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.

You cannot set the database isolation level on IBM Informix SE. ♦

ANSI



SE

Checking the Result of a FETCH

You can check the result of each FETCH statement in the SQLCODE field of the SQLCA. The variable you check is shown in the following table:

4GL	ESQL/C	ESQL/COBOL
SQLCA.SQLCODE STATUS	sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA

If a row is returned successfully, SQLCODE is set to zero. If no row is found, the FETCH statement sets the return code to 100 to indicate “row not found,” and the current row is unchanged. Five conditions set the variable value to 100, indicating “row not found,” as follows:

- 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.

References

In this manual, for further information about using the FETCH statement with dynamic management statements, see the following statements: ALLOCATE DESCRIPTOR, CLOSE, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, GET DESCRIPTOR, OPEN, PREPARE, and SET DESCRIPTOR.

Also in this manual, for further information about error checking, see [Chapter 5, “Error Handling with SQLCA.”](#) For further information about the system descriptor area, see [Chapter 6, “Using Descriptors.”](#)

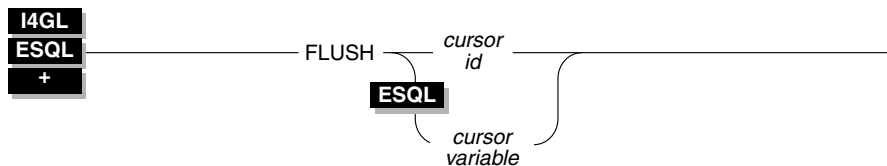
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of the FETCH statement.

FLUSH

Purpose

Use the FLUSH statement to force rows that were buffered by a PUT statement to be written to the database.

Syntax



cursor id is the identifier of a cursor that is associated with an INSERT statement.

cursor variable is an embedded variable name that identifies a cursor that is associated with an INSERT statement.

Usage

The PUT statement adds a row to a buffer and the buffer is written to the database when it is filled. Use the FLUSH statement to force the insertion even 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.

An example of a FLUSH statement follows:

```
FLUSH icurs
```


Error Checking FLUSH Statements

The SQLCA 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, as shown in the following table:

4GL	ESQL/C	ESQL/COBOL
STATUS SQLCA.SQLCODE	sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA
SQLCA.SQLERRD[3]	sqlca.sqlerrd[2]	SQLERRD[3] OF SQLCA

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 only discovered when the buffer is flushed. When an error is discovered, rows in the buffer 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, while SQLERRD contains the number of rows that were successfully inserted. (Uninserted rows are discarded from the buffer.)

Counting Total and Pending Rows

To count the number of rows actually inserted into the database, as well as the number not yet inserted, follow these steps:

1. Prepare two integer variables, for example, **total** and **pending**.
2. When the cursor is opened, set both variables to zero.
3. Each time a PUT statement is executed, increment both **total** and **pending**.
4. Whenever a PUT or FLUSH statement is executed, or the cursor is closed, subtract the third field of the SQLERRD array from **pending**.

References

In this manual, see the following statements: CLOSE, DECLARE, OPEN, and PUT.

Also in this manual, for information about SQLCA, see [Chapter 5, “Error Handling with SQLCA.”](#)

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of FLUSH.

FREE

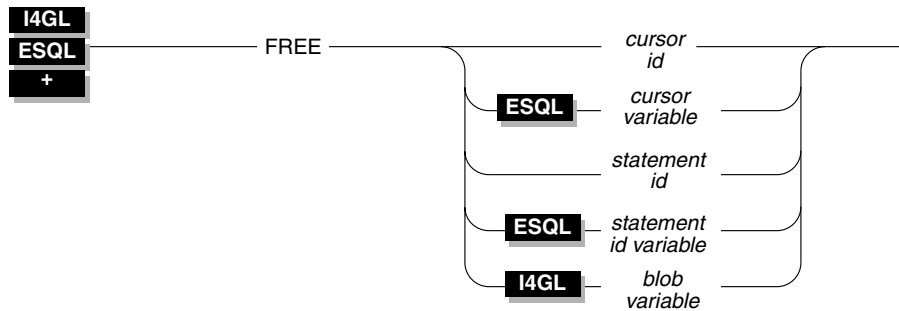
Purpose

The FREE statement releases resources that are allocated to a prepared statement or to a cursor.

The FREE statement has the additional ability to release a program variable of the TEXT or BYTE data type. ♦

I4GL

Syntax



blob variable is the name of an IBM Informix 4GL program variable of the TEXT or BYTE data type.

cursor id is the identifier of a cursor that you declared for a SELECT or INSERT statement.

cursor variable is an embedded variable name that identifies a cursor that you declared for a SELECT or INSERT statement.

statement id is the identifier of an SQL statement that you prepared with the PREPARE statement.

statement id variable is an embedded variable name that identifies an SQL statement that you prepared with the PREPARE statement.

Usage

The FREE statement releases the resources allocated for a prepared statement or a declared cursor in the application development tool and the database server. Resources are allocated when you prepare a statement or when you open a cursor (see the DECLARE and OPEN statements on pages [7-107](#) and [7-207](#), respectively.)

The sum of the number of open cursors and the number of prepared statements that you can have at one time, in one process, is limited by the amount of free memory available in the system. Use FREE *statement id* or FREE *statement id variable* to release the resources held by a prepared statement; use FREE *cursor id* or FREE *cursor variable* to release resources held by a cursor.

Freeing a Statement

If you prepared a statement (but did not declare a cursor for it), `FREE statement id` (or *statement id variable*) releases the resources in both the application development tool and the database server.

If you declared a cursor for a prepared statement, `FREE statement id` (or *statement id variable*) only releases the resources in the application development tool; the cursor still can be used. The resources in the database server are released only when you free the cursor.

After freeing a statement, you cannot execute it or declare a cursor for it until you prepare it again.

The following IBM Informix 4GL code shows the sequence of statements used to free an implicitly prepared statement.

Figure 7-38

Freeing an implicitly prepared statement in IBM Informix 4GL

```
DECLARE s_curs CURSOR FOR SELECT * FROM orders
OPEN s_curs
CLOSE s_curs
FREE s_curs
```

The following IBM Informix 4GL example shows the sequence of statements used to release the resources of an explicitly prepared statement.

Figure 7-39

Freeing an explicitly prepared statement in IBM Informix 4GL

```
PREPARE sel_stmt FROM
    "SELECT * FROM customer ",
    "WHERE customer_num BETWEEN 100 AND 200"
DECLARE sel_curs CURSOR FOR sel_stmt
OPEN sel_curs
.
.
.
CLOSE sel_curs
FREE sel_stmt
FREE sel_curs
```

Freeing a Cursor

If you declared a cursor for a prepared statement, freeing the cursor releases only the resources in the database server. To release the resources for the statement in the application development tool, use `FREE statement id` (or `statement id variable`).

If a cursor is not declared for a prepared statement, freeing the cursor releases the resources in both the application development tool and the database server.

After a cursor is freed, it cannot be opened until it is declared again. It is recommended that the cursor be explicitly closed before it is freed.

Freeing BLOB Storage with IBM Informix 4GL

4GL

If you use a `FREE` statement with a `BYTE` or `TEXT` variable stored in memory, the `FREE` statement releases all memory associated with the variable and renders the variable unusable. You must reinitialize the variable using the `LOCATE` statement before you can use it again.

If you declare a cursor and fetch a `TEXT` or `BYTE` variable into memory, you must free the variable as well as the cursor. Using the `FREE` statement on both the variable and cursor releases resources dedicated to each. However, if you use a `BYTE` or `TEXT` variable inside a function and the variable is local to the function, the memory is freed when the function is exited.

If you use a `FREE` statement with a `BYTE` or `TEXT` variable stored in a file, the `FREE` statement deletes the file from the host operating system and renders the variable unusable. You must reinitialize the variable using the `LOCATE` statement before you can use it again. ♦

References

In this manual, see the following statements: `CLOSE`, `DECLARE`, `EXECUTE`, `EXECUTE IMMEDIATE`, and `PREPARE`.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of the `FREE` statement.

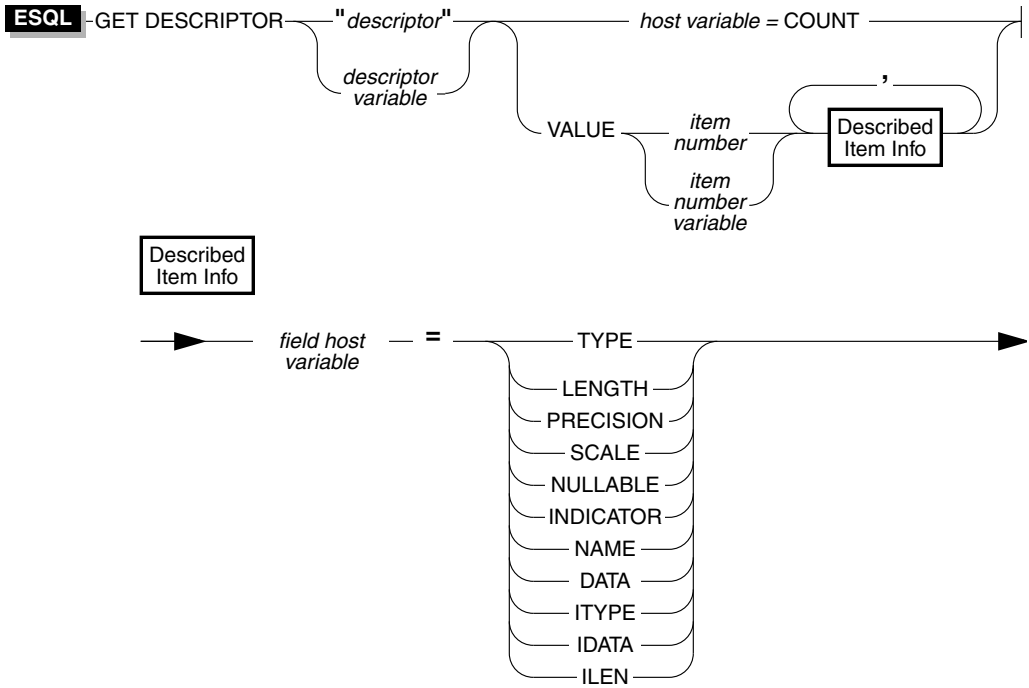
GET DESCRIPTOR

Purpose

Use the GET DESCRIPTOR statement to accomplish three separate tasks:

- Determine how many values are described in a system descriptor area by retrieving the value in the COUNT field
- Determine characteristics of each of the columns or expressions described in the system descriptor area
- Copy a value out of the system descriptor area and into a host variable after a FETCH statement

Syntax



- descriptor* is a quoted string that identifies a system descriptor area that is already allocated.
- descriptor variable* is an embedded variable name that identifies a system descriptor area that is already allocated.
- field host variable* is the name of a host variable that receives the contents of the indicated field of the system descriptor area. The field host variable must be of an appropriate type to receive the value from the system descriptor area.
- host variable* is the name of an integer host variable.

item number is an unsigned integer that represents one of the values in the descriptor area.

item number variable is the name of an integer variable that contains an unsigned integer that represents one of the values in the descriptor area.

Usage

If an error occurs during the assignment to any of the identified host variables, the contents of the host variable is undefined.

The role and contents of each of the fields in a system descriptor area are described in [Chapter 6](#).

The host variables used in the GET DESCRIPTOR statement must be declared in the BEGIN DECLARE SECTION of an ESQL program. See your embedded-language manual for specifics.

Using the *COUNT* Keyword

Use the COUNT keyword to determine how many values are described into the system descriptor area.

The following IBM Informix ESQL/C example shows how to use a GET DESCRIPTOR statement with a host variable to determine how many values are described in the system descriptor area called **desc1**.

```
main()
{
  $int h_type, h_count;
  $ALLOCATE DESCRIPTOR 'desc1' WITH MAX OCCURENCES 20;

  /* This section of program would prepare a SELECT or INSERT *
   * statement int the s_id statement id.
   */
  $DESCRIBE s_id USING SQL DESCRIPTOR 'desc1';

  $GET DESCRIPTOR 'desc1' $h_count = COUNT;
  ...
}
```

VALUE Clause

Use the VALUE clause to obtain information about a described column or expression or to retrieve values returned by the database server.

The *item number* must be greater than zero and less than the number of occurrences specified when the system descriptor area was allocated using ALLOCATE DESCRIPTOR.

Using the VALUE Clause After a Describe

After you describe a SELECT or INSERT statement, the characteristics of each of the columns or expressions in the select list of the SELECT statement or the characteristics of each of the columns in the INSERT statement are returned in the system descriptor area. Each of the values in the system descriptor area describes one returned column or expression. Each of the fields, and its possible contents, are described in [Chapter 6](#).

The following IBM Informix ESQL/C example shows how a GET DESCRIPTOR statement can be used to obtain data type information from the **demodesc** system descriptor area.

Figure 7-40

A program fragment that copies data type information into host variables for later analysis

```
$ 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 returned by the database server into the TYPE field is a defined integer. You can evaluate the type returned by testing for a specific integer value. The codes for the TYPE field are listed in [Chapter 6](#).

X/O

In X/Open mode, the X/Open code is returned to the TYPE field. You must be careful not to mix the two modes because errors can result. For example, if a particular type is not defined under X/Open mode but is defined for IBM Informix products, the execution of 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 types 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, use a GET DESCRIPTOR statement after each fetch for each of the values in the select list. If there are three values 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 IBM Informix ESQL/C example shows how you can copy the data out of the DATA field into a host variable (**result**) after a fetch. For this example, it is predetermined that all values returned are the same data type.

Figure 7-41

An ESQL/C program fragment that copies values from the DATA field into a host variable

```
$FETCH democursor USING SQL DESCRIPTOR 'demodesc';
if (sqlca.sqlcode != 0) break;
for (i = 1; i <= desc_count; i++)
{
    $ GET DESCRIPTOR 'demodesc' VALUE $i $result = DATA;
    printf("%s ", result);
}
printf("\n");
}
```

The following IBM Informix ESQL/COBOL example shows how you can copy the data out of the DATA field into host variables after a fetch. The first use of GET DESCRIPTOR uses a literal item number; the second GET DESCRIPTOR uses a host variable to hold the item number.

Figure 7-42

GET DESCRIPTOR after a fetch in ESQL/COBOL

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 COUNTSQLINT.
01 ITEMNSQLINT.
01 TYPESQLINT.
01 LENGTHSQLINT.
01 LONGVALSQLINT.
01 CHVALSQLCHAR(21).
EXEC SQL END DECLARE SECTION END-EXEC.
```

```
EXEC SQL GET DESCRIPTOR 'desc1' VALUE 1
      :TYPE = TYPE, :LENGTH = LENGTH, :CHVAL = DATA
      END-EXEC.

MOVE 2 TO ITEMNO.
EXEC SQL GET DESCRIPTOR 'desc1' VALUE :ITEMNO
      :TYPE = TYPE, :LONGVAL = DATA
      END-EXEC.

.
.
.
```

Fetching a Null Value

When you use GET DESCRIPTOR after a fetch and the value fetched is null, then 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.

References

In this manual, for further information about using dynamic SQL statements, see the following statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, OPEN, PREPARE, PUT, and SET DESCRIPTOR.

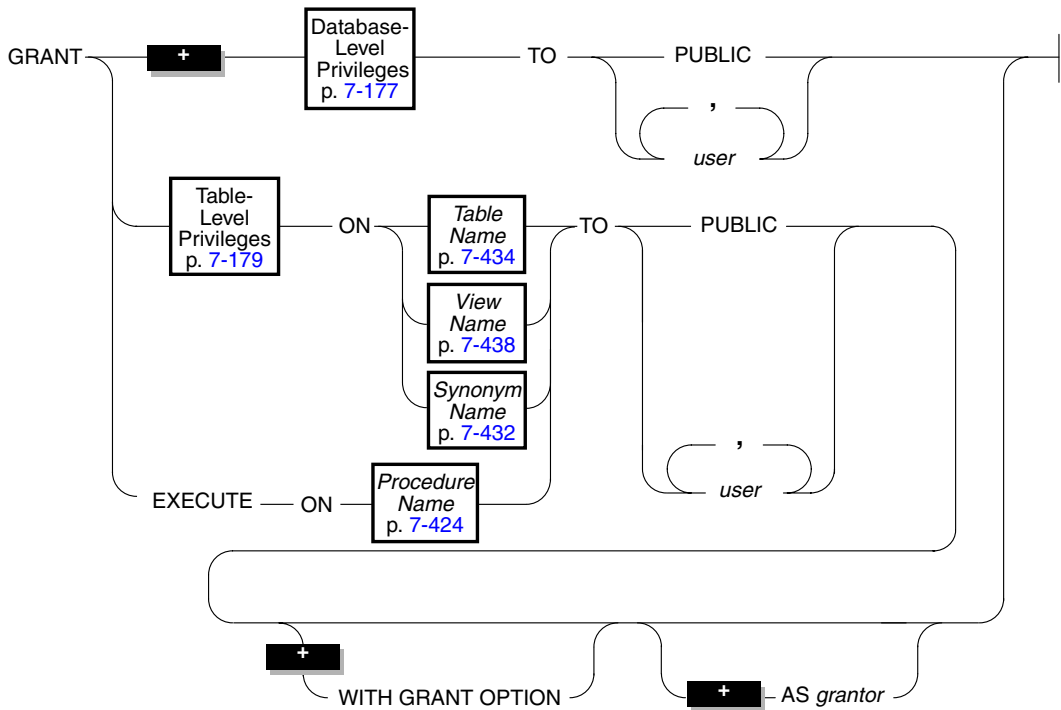
Also in this manual, for further information about the system descriptor area, see [Chapter 6, "Using Descriptors."](#)

GRANT

Purpose

Use the GRANT statement to specify access privileges for a database or for the tables and views in a database.

Syntax



<i>grantor</i>	identifies the user who is granting the privilege to user. As the current user, you are the default grantor.
<i>user</i>	names the user or users who receive privileges. Granting privileges to PUBLIC extends a privilege to the class of all authorized users, both current and future.

Usage

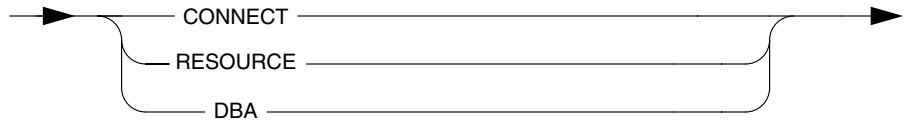
A GRANT statement can extend user privileges but cannot limit existing privileges. Later GRANT statements do not affect privileges already granted to a user. When database-level privileges collide with table-level privileges, the more restrictive privileges take precedence. You can grant table-level 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 granted.

SE

You cannot use a ROLLBACK WORK statement to undo a GRANT statement that successfully executes. If you roll back a transaction that contains a GRANT statement, the privilege is not revoked and you do not receive an error message. ♦

Database-Level Privileges



When you create a database, you alone have access to it. The database remains inaccessible to other users until you, as database administrator (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:

- CONNECT** Connect privilege gives you the ability to query and modify data. You can modify the database schema if you own the object you wish to modify. Any user with Connect privilege can perform the following functions:
- Execute `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements, provided the user has the necessary table-level privileges
 - Create views, provided the user has 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` keywords)

RESOURCE	<p>Resource privilege 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:</p> <ul style="list-style-type: none">■ Create new tables■ Create new indexes■ Create new procedures
DBA	<p>In addition to the capabilities of the Resource privilege, the holder of the DBA privilege can perform the following functions:</p> <ul style="list-style-type: none">■ Grant any database-level privilege, including DBA privilege, to another user■ Use the NEXT SIZE keywords to alter extent sizes in the system catalog■ Insert, delete, or update rows of any system catalog table except systables■ Drop any object, regardless of who owns it■ Create tables, views, and indexes, and specify another user as owner of the objects■ Execute the DROP DATABASE statements■ Execute the START DATABASE and ROLLFORWARD DATABASE statements ♦

SE



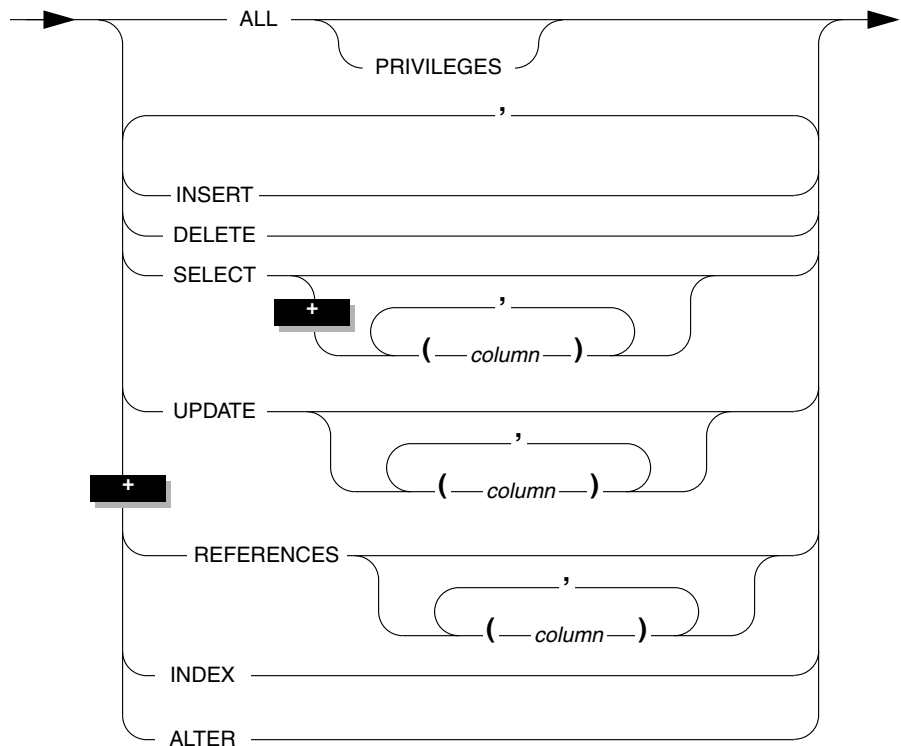
User **informix** has the privilege required to alter tables in the system catalog, including the **systables** table.

Important: Although user **informix** can modify the system catalog tables, it is strongly recommended that you do not update, delete, or alter any rows in these tables. Modifying the system catalog tables can destroy the integrity of the database.

The following example uses the PUBLIC keyword to grant Connect privilege on the **stores5** database to all users:

```
GRANT CONNECT ON stores5 TO PUBLIC
```


Table-Level Privileges



As owner of a table, or as 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 by virtue of ownership. 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 Select, Update, or References privilege, you can limit the privileges by listing the names of specific columns.

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 Resource privilege for the database to modify the database structure. The table-level privileges are defined as follows:

SELECT	Ability to name any column in SELECT statements. You can restrict the Select privilege to one or more columns by listing them.
UPDATE	Ability to name any column in UPDATE statements. You can restrict the Update privilege to one or more columns by listing them.
INSERT	Ability to insert rows.
DELETE	Ability to delete rows.
INDEX	Ability to create permanent indexes. You must have Resource privilege to take advantage of Index privilege. (Any user with Connect privilege can create an index on temporary tables.)
ALTER	Ability to add or delete columns, modify column data types, or add or delete constraints.
REFERENCES	Ability to reference columns in referential constraints. You must have Resource privilege to take advantage of References privilege. (However, you can add a referential constraint during an ALTER TABLE statement. This does not require that you have Resource privilege on the database.) You can restrict the References privilege to one or more columns by listing them.
ALL	All privileges. The PRIVILEGES keyword is optional.

The following example grants Delete and Select privileges on all columns, and Update privilege on **customer_num**, **fname**, and **lname** for the **customer** table, to users **mary** and **john**:

```
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 follows:

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
  ON customer TO PUBLIC
```

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 shown in the following example:

```
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. ♦

Stored Procedure Privileges

Use the `EXECUTE ON` option with a procedure name to grant another user the ability to run a stored procedure that you own.

When you create an owner-privileged stored procedure, the default privilege is `PUBLIC`.

ANSI

If you create a procedure in a database that is ANSI-compliant, no default-level privileges are granted. ♦

WITH GRANT OPTION

Using the `WITH GRANT OPTION` keywords 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*, and to whomever *user* conveys the right to grant privileges. If you use the `WITH GRANT OPTION` keywords, you can no longer control the dissemination of privileges.

If you revoke from *user* the privilege that you granted using the WITH GRANT OPTION keywords, you sever the chain of privileges. That is, when you revoke privileges from *user*, you revoke automatically 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 table **items**, issue these statements to grant access only to Mary:

```
REVOKE ALL ON items FROM PUBLIC
GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION
```

Mary uses her new authority to grant both 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 Mary's access privileges on **items**:

```
REVOKE SELECT, UPDATE ON items FROM mary
```

This single statement effectively revokes all privileges on **items** from Mary, Cathy, and Paul.

If you want to create a chain of privileges with some other user as the source of the privilege, use the AS *grantor* clause.

AS grantor

The AS *grantor* clause enables you to establish a chain of privileges with another user as the source of the privileges. In so doing, you relinquish 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 table **items** and you grant all privileges to Tom, along with the right to grant all privileges:

```
REVOKE ALL ON items FROM PUBLIC
GRANT ALL ON items TO tom WITH GRANT OPTION
```

You also grant Select and Update privileges to Jim, but you specify that the grant is made as Tom. (This means that the records of the database server will show that Tom is the grantor of the grant in the **sysstabauth** system catalog table, rather than you.)

```
GRANT SELECT, UPDATE ON items TO jim AS tom
```

Later, you decide to revoke Tom's privileges on **items**; you issue the following statement:

```
REVOKE ALL ON items FROM tom
```

When you try to revoke Jim's privileges with a similar statement, however, the database server returns an error:

```
REVOKE SELECT, UPDATE ON items FROM jim
```

```
580: Cannot revoke permission.
```

Because the database server record shows the original grantor as Tom, you cannot revoke the privilege. Even though 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 creating a view, if you do not own the underlying tables, you must have at least 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 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 (WITH GRANT OPTION). If you are able to create the view, you can grant at least Select privilege on it. If a view is built on more than one table, the view only has Select privilege. You cannot grant Index, Alter, or References privileges on a view (or All privilege, since All includes Index and Alter).

For views that reference only tables in the current database, if the owner of a view loses Select privilege on any of the tables underlying the view, the view is dropped.

For detailed information, refer to the CREATE TABLE statement, which also describes creating views.

References

In this manual, see the following statements: CREATE TABLE and REVOKE.

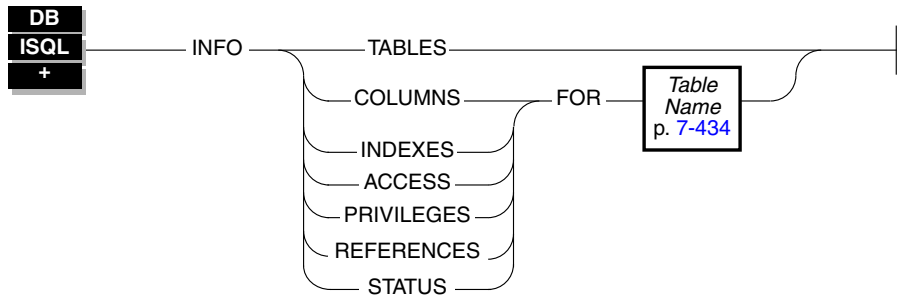
In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of database and table-level privileges, and privileges and security.

INFO

Purpose

Use the INFO statement to display a variety of information about databases and tables.

Syntax



Usage

The following types of information can be displayed by issuing the INFO statement:

- The names of the tables in the current database
- Column information for a specified table
- Index information for a specified table
- Access privileges for a specified table
- References privileges for the columns of a specified table
- Status information for a specified table

Instead of using the SQL statement INFO, you can use the **Info** options on the SQL menu or TABLE menu to display the same information.

Displaying Tables, Columns, and Indexes

You can use keywords in your INFO statement to display a list of tables, information about the columns of a table, or information about the indexes of a table.

Use the TABLES keyword to display a list of the tables in the current database. The name of a table can appear in one of two 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 table name is preceded by the owner's name, as in **"june".cust_calls**.

Use the COLUMNS FOR keywords to display the names and data types of the columns in a specified table and whether null values are allowed. The following examples show an INFO statement and the resulting display of information about the columns in a table.

Figure 7-43

INFO statement requesting column information

```
INFO COLUMNS FOR cust_calls
```

Figure 7-44

Display of column information

Column name	Type	Nulls
customer_num	INTEGER	no
call_dtime	DATETIME YEAR TO MINUTE	yes
user_id	CHAR(18)	yes
call_code	CHAR(1)	yes
call_descr	CHAR(240)	yes
res_dtime	DATETIME YEAR TO MINUTE	yes
res_descr	CHAR(240)	yes

Use the INDEXES FOR keywords 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. The following examples show an INFO statement and the resulting display of information about the indexes of a table.

Figure 7-45
INFO statement requesting index information

```
INFO INDEXES FOR cust_calls
```

Figure 7-46
Display of index information

Index name	Owner	Type	Cluster	Columns
c_num_dt_ix	velma	unique	No	customer_num call_dtime
c_num_cus_ix	velma	dupls	No	customer_num

Displaying Privileges, References, and Status

You can use keywords in your INFO statement to display information about the access privileges (including References privilege) or status of a table.

Use the ACCESS FOR or PRIVILEGES FOR keywords to display six of the user access privileges for a specified table. The following examples show an INFO statement and the resulting display of user privileges for a table.

Figure 7-47
INFO statement requesting privileges information

```
INFO PRIVILEGES FOR cust_calls
```

Figure 7-48
Display of privileges information

User	Select	Update	Insert	Delete	Index	Alter
public	All	All	Yes	Yes	Yes	No

Use the REFERENCES FOR keywords to display the References privilege for users for the columns of a specified table. The following examples show an INFO statement and the resulting display.

Figure 7-49
INFO statement requesting References privilege information

```
INFO REFERENCES FOR newtable
```

Figure 7-50*Display of References privilege information*

User	Column References
betty	col1 col2 col3
wilma	All
public	None

The output indicates that the user “betty” can reference columns col1, col2, and col3 of the specified table, the user “wilma” can reference all the columns in the table, and “public” cannot access any of the columns in the table.

If you want information about database-level privileges, you must use a SELECT statement to access the **sysusers** system catalog table.

See the GRANT and REVOKE statements for more information about database and table access privileges.

Use the STATUS FOR keywords to display information about the owner, row length, number of rows and columns, creation date, and status of audit trails for a specified table. The following examples show an INFO statement and the resulting display of status information for a table on an IBM Informix SE database server.

Figure 7-51*INFO statement requesting status information*

```
INFO STATUS FOR cust_calls
```

Figure 7-52*Display of status information*

Table Name	cust_calls
Owner	velma
Row Size	517
Number of Rows	7
Number of Columns	7
Date Created	01/28/1991
Audit Trail File	

The audit trail file line does not appear for tables on IBM Informix OnLine. ♦

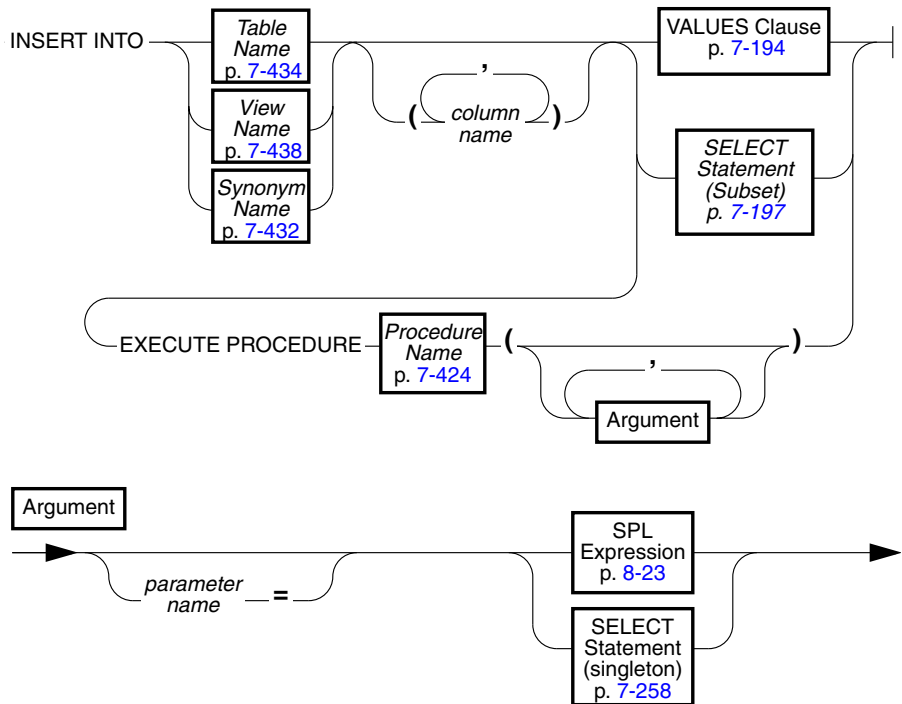
OL

INSERT

Purpose

Use the INSERT statement to insert one or more new rows into a table or view.

Syntax



column name is the column that receives the new data.

parameter name is the name of the parameter as defined by its CREATE PROCEDURE statement.

Usage

You can 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 Insert privilege for the table (see the GRANT statement on page 7-175). 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” later in this section.

If you insert data into a table that has data integrity constraints associated with it, the data inserted must meet the constraint criteria. If it does not, the database server returns an error.

Specifying Columns

If you do not explicitly specify one or more columns, data is inserted into columns using the column order of the table established when the table was created or last altered. The column order is listed in the **syscolumns** system catalog table.

E/C

You can use the DESCRIBE statement with a SELECT statement to obtain the column order or the data type of the columns in a table. (For more information about the DESCRIBE statement, see page [7-125](#).) ♦

The number of columns specified in the INSERT INTO clause must equal the number of values supplied in the VALUES clause, or supplied 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 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 (page 7-100).

If several users are entering sensitive information into a single table, you can use the USER function to limit their view to only the specific rows that each inserted. The following example contains a view and an INSERT statement that achieve this effect:

```
CREATE VIEW salary_view AS
  SELECT lname, fname, current_salary
     FROM salary
     WHERE entered_by = USER

INSERT INTO salary
  VALUES ("Smith", "Pat", 75000, USER)
```

Inserting Rows with a Cursor

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 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
- An OPEN statement implicitly closes and then reopens the cursor
- A COMMIT WORK statement ends the transaction

When the insert buffer is flushed, the front-end 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 following the last successfully inserted rows are discarded. ♦

I4GL

ESQL

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. 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, you can undo the insertion using the ROLLBACK WORK statement. 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

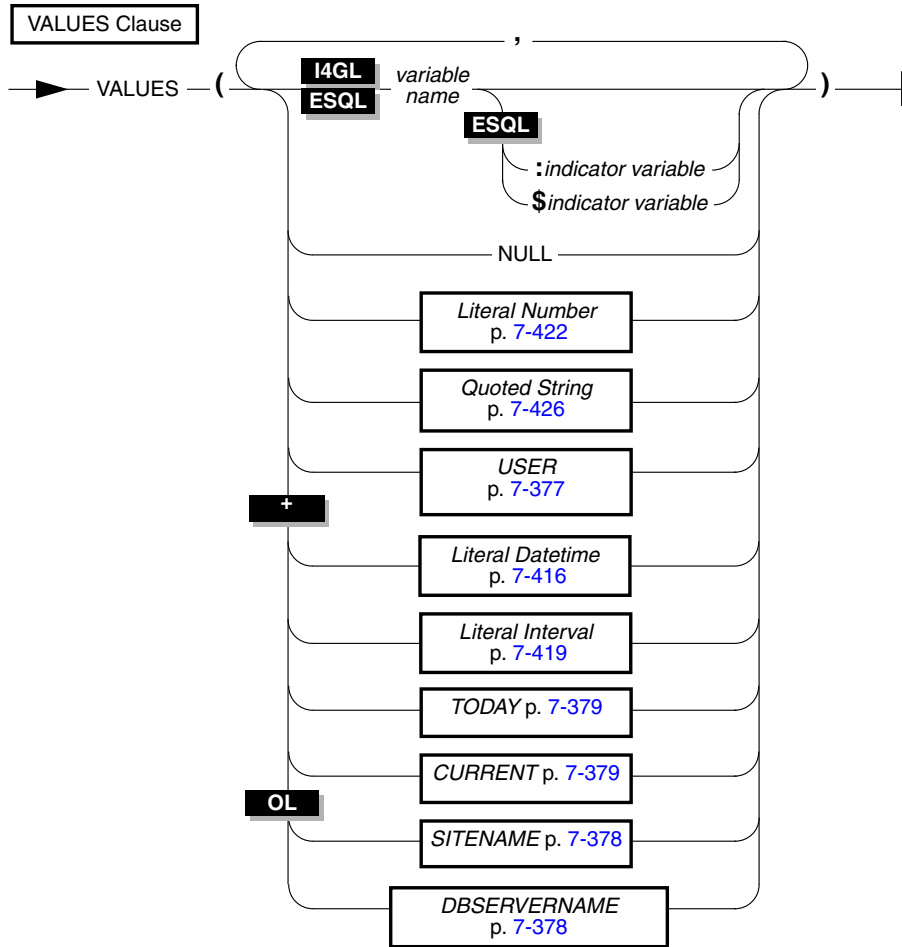
If you are inserting rows into an ANSI-compliant database, transactions are implicit and all database modifications take place within a transaction. In this case, if an INSERT statement fails, you can use the ROLLBACK WORK statement to undo the insertions. ♦

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 ROLLBACK WORK statement, where none of the modifications are made to the database. If the number of rows affected by a *single* INSERT statement is quite large, 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.

SE

To prevent this situation, either insert fewer rows per transaction or lock the entire table before you execute the INSERT statement. ♦

VALUES Clause



indicator variable is a variable associated with a host variable that indicates when an ESQL statement returns a null value to that host variable.

variable name is a program variable, host variable, or record name as defined in an IBM Informix 4GL program or Informix embedded-language program.

I4GL

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 previously defined a RECORD-type program variable for the table, you can use the program variable in place of a list of values. ♦

If you are inserting a quoted string into a column, the maximum length of the string is 256 bytes. If you insert a value greater than 256, the database server returns an error.

I4GL

ESQL

If you are using variables, you can insert quoted strings longer than 256 bytes into a table. ♦

Value and Column Type Compatibility

Although the values you insert do not have to be the same data type as the columns receiving them, the value type and column type must be compatible. You can insert only characters into CHAR columns and only numbers or characters representing 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 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 first verifying 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 [Chapter 3, "Data Types."](#)

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 OnLine 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 containing the login account name of the current user. The SITENAME or DBSERVERNAME keyword returns the database server name on which 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")
```

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 and for all columns not listed explicitly that do not have default values associated with them. You also can use the keyword NULL 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, while all other columns of the **orders** table *not* explicitly listed in the INSERT INTO clause are also filled with null values.

Subset of SELECT Statement

You can insert the rows of data that result from a SELECT statement into a table if the insert data is selected from another table or tables. The following SELECT clauses are not supported:

- INTO TEMP clause
- ORDER BY clause

In addition, the FROM clause of the SELECT statement cannot contain the same table name as the table into which you are inserting rows, as shown in the following example:

```
INSERT INTO newtable
  SELECT item_num, order_num, quantity, stock_num,
         manu_code, total_price
  FROM items
```

Detailed information on SELECT statement syntax is provided on page [7-258](#).

Using INSERT as a Dynamic Management Statement

You can use the INSERT statement to handle situations in which 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 in the manual for your IBM Informix embedded-language product. ♦

Inserting Data Using a Stored Procedure

You can insert the rows of data that result from a procedure call into a table.

The values returned by the procedure must match those expected by the column-list in number and data type. The number and data types of the columns must match those expected by the column-list.

References

In this manual, for general use information, see the following statement: SELECT.

Also in this manual, for specific information about dynamic management statements, see the following statements: DECLARE, DESCRIBE, EXECUTE, FLUSH, OPEN, PREPARE, and PUT.

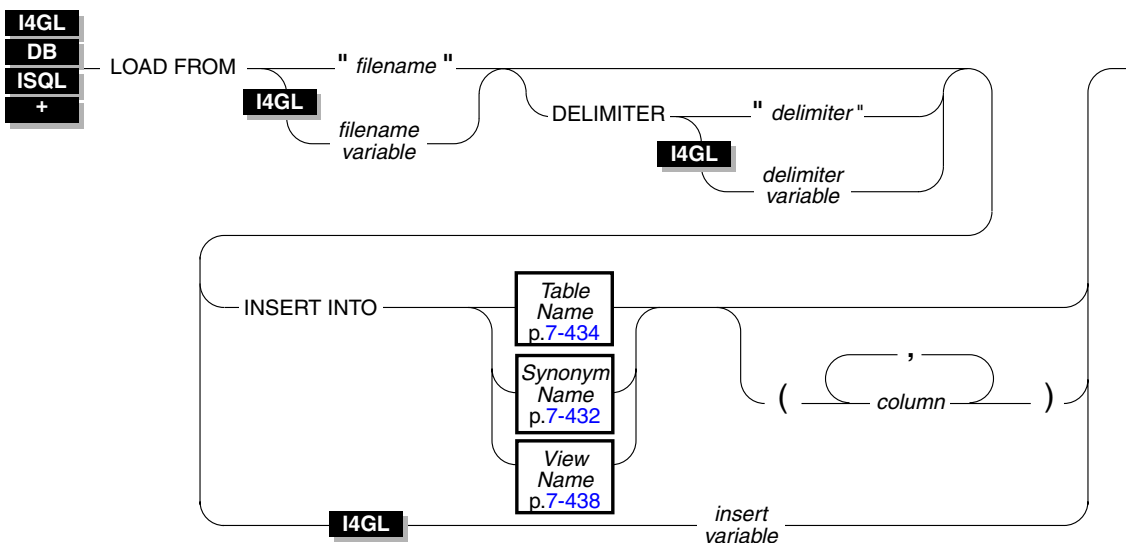
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of inserting data.

LOAD

Purpose

Use the LOAD statement to insert data from an ASCII operating system file into an existing table, synonym, or view.

Syntax



column is a column belonging to *Table Name*, *Synonym Name*, or *View Name*. You must specify column names if you are not loading data into all columns in the table, synonym, or view.

delimiter is a quoted string constant that contains the character to use as the delimiting character in the load file.

delimiter variable is a character variable that contains the character to use as the delimiter between fields. The default delimiter is the vertical bar (| = ASCII 124).

- filename* is a quoted string constant or character string that specifies the file that contains the data to load. It includes the pathname of an operating system file.
- filename variable* is a CHARACTER or VARCHAR variable that specifies the name of the file that contains the data to load.
- insert variable* is a CHARACTER or VARCHAR variable that contains an INSERT statement. IBM Informix 4GL uses the INSERT statement to determine the tables and columns in which to load the new data. The INSERT statement cannot contain a VALUES or SELECT clause. (See the INSERT statement on page [7-189](#).)

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 into which you want to insert the data. For information on database-level and table-level privileges, see the GRANT statement on page [7-175](#).

If your database has transactions but is not ANSI-compliant, you must issue a BEGIN WORK statement before using the LOAD statement.

You cannot use the PREPARE statement to preprocess a LOAD statement. ♦

The LOAD FROM File

The LOAD FROM file is the file that contains the data to add to a table. You can use the file created by the UNLOAD statement 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 specified for the table in number, order, and 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 specified for the corresponding column. Specify only values that IBM Informix 4GL, IBM Informix SQL, or DB-Access can convert to the data type of the corresponding column. The following table indicates how your IBM Informix product expects you to represent the data types in the LOAD FROM file.

Figure 7-53
Types of data and their input format for a LOAD statement

Type of Data	Expected 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	A character string in the following format: <i>month/day/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. The value must be an actual date; for example, February 30 is illegal.
MONEY	A value that can have leading currency symbols.
NULL	Nothing between the delimiters.
time	A 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 a two-digit number.

If you include any of the following special characters as part of the value of a field, you must precede the character with a backslash (\):

- Backslash
- Delimiter
- New line anywhere in the value for a VARCHAR column
- New line 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 command that the next character is to be interpreted as part of the data.

The fields corresponding to character columns can contain more characters than the defined maximum for the field. IBM Informix 4GL, IBM Informix SQL, and DB-Access ignore the extra characters.

If you are loading files containing VARCHAR or BLOB data types, note the following information:

- If you give the LOAD statement data in which the character (including VARCHAR) fields 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 characters as delimiting characters in the LOAD FROM file: 0-9, a-f, A-F, space, tab, backslash.

For more information about the format of the input file, see the discussion of the **dbload** utility in either the *IBM Informix OnLine Administrator's Guide* or the *IBM Informix SE Administrator's Guide*.

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 default delimiter character, the vertical bar (|)
- Assigns null values to the **phone** field for the first row and the **address2** field for the second row

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. If you omit this clause, IBM Informix 4GL, IBM Informix SQL, and DB-Access check the **DBDELIMITER** environment variable.

If the **DBDELIMITER** variable has not been set, the default delimiter is the vertical bar (| = ASCII 124). See Chapter 4 in this manual for information about how to set the **DBDELIMITER** environment variable.

The following statement identifies the semicolon (;) as the delimiting character:

```
LOAD FROM "/a/data/ord.loadfile" DELIMITER ";"
      INSERT INTO orders
```

INSERT INTO Clause

Use the INSERT INTO clause to specify the table, synonym, or view in which to load the new data. (See the discussion of Synonym Name, Table Name, and View Name beginning on page [7-432](#) for details.)

You must specify the column names only if one of the following conditions is true:

- You are not loading data into all columns.
- The input file does not match the default order of the columns (determined when the table was created).

The following example identifies the **price** and **discount** columns as the only columns in which to add data:

```
LOAD FROM "/tmp/prices" DELIMITER ", "
      INSERT INTO norman.worktab(price, discount)
```

References

In this manual, see the following statements: UNLOAD and INSERT.

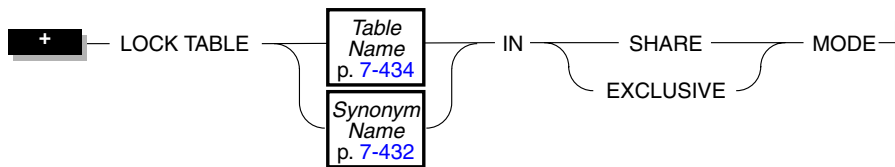
In the *IBM Informix OnLine Administrator's Guide* or the *IBM Informix SE Administrator's Guide*, see the discussion of the **dbload** utility.

LOCK TABLE

Purpose

Use the LOCK TABLE statement to control access to a table by other processes.

Syntax



Usage

You can lock a table if you own the table or have 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, and ALTER TABLE statements.

The IBM Informix SE database server does not permit more than one user to lock a table in shared mode. ♦

SE

Databases with Transactions

If your database was created with transactions, the LOCK TABLE statement succeeds only if it is executed 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 catalogs.
- 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 prevent a situation in which you exceed the maximum number of locks defined in the IBM Informix OnLine configuration.

SE

The maximum number of locks allowed by the IBM Informix SE database server is a characteristic of the particular operating system on which your database server is running. ♦

- 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 created without transactions, table locks set 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

In this manual, see the following statements: BEGIN WORK, SET ISOLATION, SET LOCK MODE, COMMIT WORK, ROLLBACK WORK, and UNLOCK TABLE.

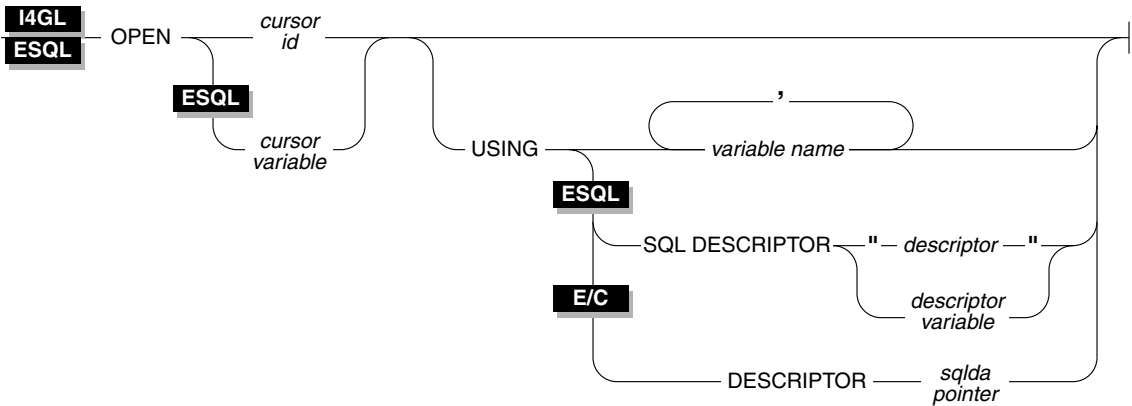
In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of locks.

OPEN

Purpose

Use the OPEN statement to activate a cursor associated with a SELECT or INSERT statement, and thereby begin execution of the SELECT or INSERT statement.

Syntax



cursor id identifies a cursor that was created by an earlier DECLARE statement.

cursor variable is an embedded-language variable that identifies a cursor that was created by an earlier DECLARE statement.

descriptor is a quoted string that identifies the system descriptor area that was previously allocated.

descriptor variable is an embedded-language variable name that identifies the system descriptor area that was previously allocated.

sqllda pointer points to an **sqllda** structure that defines the type and memory location of values that correspond to the question mark (?) placeholder in a prepared statement.

variable name is a program or host variable whose contents are to replace a ? placeholder in a prepared statement.

Usage

A cursor is created and associated with a statement using the DECLARE statement (page 7-107). When the program opens the cursor, the associated SELECT or INSERT statement is passed to the database server, which begins execution. When the program has retrieved or inserted all the rows it needs, the cursor should be closed using the CLOSE statement.

The specific actions taken by the database server differ, depending on whether the cursor is associated with a SELECT statement or an INSERT statement.

The (SELECT, INSERT, EXECUTE PROCEDURE) statement associated with a cursor is prepared implicitly by the OPEN statement. The total number of prepared objects and open cursors allowed in one program at any time is limited by the available memory. You can use the FREE statement (to free the cursor) to release the database server resources.

An error code is returned if you open a cursor that is already open. ♦

ANSI

Opening a SELECT Cursor

When you open either a select cursor or an update cursor, the SELECT statement is passed to the database server along with any values specified in the USING clause. (If the statement was previously prepared, the statement was 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.

Since this is the first time that the database server sees the query, it is the time when many errors are detected. The database server does not actually return the first row of data, but it sets a return code in the `SQLCODE` field of the `SQLCA`. The name of the field in each product is indicated in the following table:

4GL	ESQL/C	ESQL/COBOL
SQLCA.SQLCODE STATUS	sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA

The return code value is either negative or zero, as follows:

negative	An error was detected in the SELECT statement.
zero	The SELECT statement is valid.

If the SELECT statement is valid but no rows match its criteria, the first FETCH statement returns a value of 100 (SQLNOTFOUND), end of data.

The following example illustrates a simple OPEN statement in IBM Informix 4GL:

```
DECLARE s_curs CURSOR FOR
  SELECT * FROM orders
OPEN s_curs
```

If you are working in a database with explicit transactions, you must open an update cursor within a transaction. This requirement is waived if you declared the cursor using the WITH HOLD keywords. (See the DECLARE statement on page [7-107](#).)

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 [7-107](#).)

An OPEN statement for a cursor associated with an INSERT statement cannot include a USING clause.

The following IBM Informix 4GL code fragment illustrates an OPEN statement with an insert cursor.

Figure 7-54

An OPEN statement with an insert cursor in IBM Informix 4GL

```
PREPARE s1 FROM
    "INSERT INTO manufact ",
    "VALUES ("NPR", "Napier)"
DECLARE in_curs CURSOR FOR s1
OPEN in_curs
PUT in_curs
CLOSE in_curs
```

Reopening a SELECT Cursor

The values 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 the program closes the open cursor, which releases the active set, or until a subsequent OPEN statement closes the cursor and reopens it.

Reopening the cursor creates a new active set based on the current values of the variables. If the program variables 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, if data in the table was modified since the previous OPEN statement, the rows in the active set can be different.

Reopening an INSERT Cursor

When you reopen an insert cursor that is already open, you effectively flush the insert buffer; any rows stored in the INSERT buffer are written into the database table. The database server first closes the cursor, which accounts for the flush, and then reopens the cursor. See the discussion of the PUT statement on page [7-230](#) for information about checking errors and counting inserted rows.

USING Clause

The USING clause is required when the cursor is associated with a prepared SELECT statement that includes ? placeholders. (See the PREPARE statement on page 7-218.) You can supply values for these parameters in one of two ways.

Naming Variables in USING

If you know the number of parameters to be supplied at run time and their data types, you can define the parameters 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 ? 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 code the SELECT statement as part of the DECLARE statement. ♦

The following IBM Informix 4GL code fragment illustrates how a program employs the USING clause and input variables to establish the search criteria in a SELECT statement. The program stores user input in a program variable named **zip** and names this variable in the USING clause. Its current value replaces the ? in the prepared SELECT statement so that the rows returned through the cursor are the ones requested by the user.

Figure 7-55

The USING keyword with the OPEN statement in IBM Informix 4GL

```
PREPARE zipsel FROM
    "SELECT * FROM customer WHERE zipcode MATCHES ?"
DECLARE q_curs CURSOR FOR zipsel
PROMPT "Enter a zipcode: " FOR zip
OPEN q_curs USING zip
```

The next example illustrates the USING clause in an IBM Informix ESQL/C code fragment.

Figure 7-56

The USING keyword with the OPEN statement in IBM Informix ESQL/C

```

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");
$prepare statement_1 from select_1;
$declare q_curs cursor for statement_1;
$open q_curs using $o_date, $c_num;
    
```

USING SQL DESCRIPTOR Clause

ESQL

You also can 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 systemdescriptor 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 **count** must be less than or equal to the value of occurrences specified when the system descriptor area was allocated.

For further information, refer to the discussion of the system descriptor area in the manual for your IBM Informix ESQL product and in [Chapter 6](#).

Figure 7-57

Sample OPEN USING SQL DESCRIPTOR clause in ESQL/C

```

$OPEN selcurs USING SQL DESCRIPTOR "desc1";
    
```

Figure 7-58

Sample OPEN USING SQL DESCRIPTOR clause in ESQL/COBOL

```

EXEC SQL OPEN SEL_CURS USING SQL DESCRIPTOR "DESC1" END-EXEC.
    
```

◆

E/C

USING DESCRIPTOR Clause

You can pass parameters for a prepared statement in the form of an `sqlda` pointer structure, which lists the data type and memory location of one or more values to replace ? placeholders. For further information, refer to the `sqlda` discussion in the *IBM Informix ESQL/C Programmer's Manual*.

Figure 7-59

Sample OPEN USING DESCRIPTOR clause in ESQL/C

```
struct sqlda *sdp;
...
$open selcurs using descriptor sdp;
◆
```

Choosing Between OPEN and FOREACH

I4GL

IBM Informix 4GL contains a FOREACH statement that performs an implied OPEN statement. You can use the FOREACH statement with a cursor associated with a SELECT statement to replace the OPEN, FETCH, and CLOSE combination of statements. The FOREACH statement is compatible with sequential, scroll, or hold cursors.

You cannot use the FOREACH statement if the OPEN statement would require a USING clause. That is, if the cursor is associated with a prepared SELECT statement that includes ? parameters, you are required to open the cursor with an OPEN statement that includes a USING clause.

An example illustrating a FOREACH loop that replaces the OPEN, FETCH, and CLOSE series of statements follows. This IBM Informix 4GL program fragment selects records from the **customer** table into the program array **p_cust**. The program assumes that fewer than 2,000 customers are listed in the table. ◆

Figure 7-60

A FOREACH loop that replaces OPEN, FETCH, and CLOSE in IBM Informix 4GL

```
GLOBALS
  DEFINE p_cust ARRAY[2000] OF RECORD
    customer_num LIKE customer.customer_num,
    fname LIKE customer.fname,
    lname LIKE customer.lname,
    company LIKE customer.company
  END RECORD
END GLOBALS

FUNCTION load_cust() { fill array p_cust, return count }
  DEFINE i SMALLINT
  DECLARE cust_cur CURSOR FOR
    SELECT customer_num, fname, lname, company
    FROM customer
  LET i = 1
  FOREACH cust_cur INTO p_cust[i].*
    LET i = i + 1
    IF i > 2000 THEN
      DISPLAY "Stopping at 2000 customers"
      EXIT FOREACH
    END IF
  END FOREACH
  RETURN i
END FUNCTION

MAIN
  DISPLAY load_cust(), "customers read from database."
END MAIN
```

The Relationship Between OPEN and FREE

The database server allocates resources to prepared statements and open cursors. If you release resources with a *FREE cursor id* or *FREE cursor variable* statement, you cannot use the cursor unless you declare the cursor again. If you execute a *FREE statement id* or *FREE statement id variable* statement, you cannot open the cursor associated with the *statement id* or *statement id variable* unless you prepare the *statement id* or *statement id variable* again.

References

In this manual, see the following cursor-related statements: CLOSE, DECLARE, and FREE. For insert cursors, see also PUT and FLUSH.

Also in this manual, for further information about dynamic SQL statements, see the following statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, PREPARE, PUT, and SET DESCRIPTOR. For further information about the system descriptor area and the **sqlda** structure, see Chapter 6 of this manual.

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of the OPEN statement.

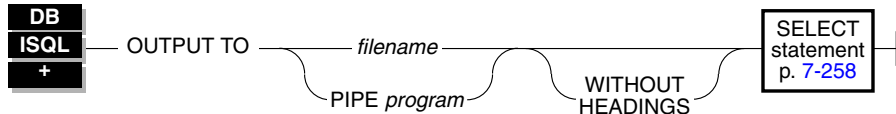
Refer also to the manual for your embedded-language product for further information about the system descriptor area and the **sqlda** structure.

OUTPUT

Purpose

Use the OUTPUT statement to send query results directly to an operating system file or to pipe it to another program.

Syntax



filename is the name of the operating system file in which you want to store the results of the query.

program is the name of the program where you want the query results piped or otherwise sent.

Usage

You can send the results of a query to an operating system file by specifying the full pathname for the file. If the file already exists, the output overwrites the current contents, as follows:

```

OUTPUT TO /usr/april/query1
  SELECT * FROM cust_calls WHERE call_code = "L"
  
```

You can display the results of a query without column headings by using the WITHOUT HEADINGS keywords, as follows:

```

OUTPUT TO /usr/april/query1
  WITHOUT HEADINGS
  SELECT * FROM cust_calls WHERE call_code = "L"
  
```

You also can use the keyword PIPE to send the query results to another program, as follows:

```
OUTPUT TO PIPE more
SELECT customer_num, call_dtime, call_code
FROM cust_calls
```

References

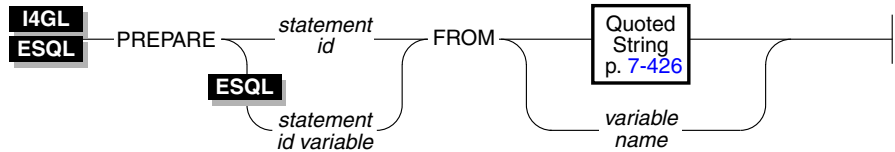
In this manual, see the following statements: SELECT and UNLOAD.

PREPARE

Purpose

Use the PREPARE statement to parse, validate, and generate an execution plan for SQL statements in an 4GL or IBM Informix ESQL program at run time.

Syntax



statement id is an SQL statement identifier. The *statement id* must conform to the same rules as any identifier, as described in the Identifier segment on page [7-399](#).

statement id variable is the name of an embedded-language character variable that contains the SQL statement identifier. The *id variable* and the identifier it contains must conform to the same rules as any identifier, as described in the Identifier segment on page [7-399](#).

variable name is a character IBM Informix 4GL program variable or an IBM Informix ESQL host variable that contains the text of the SQL statement to be prepared.

The maximum length for a quoted string in a PREPARE statement is 2048 bytes.

Usage

The PREPARE statement permits your program to assemble the text of an SQL statement at run time and make it executable. This dynamic form of SQL is accomplished in three steps:

1. A PREPARE statement accepts statement text as input, either as a quoted string or stored within a character variable. Statement text can contain question mark (?) placeholders to represent values that are to be defined when the statement is executed.
2. An EXECUTE or OPEN statement can supply the required input values and execute the prepared statement once or many times.
3. Resources allocated to the prepared statement can be released later using the FREE statement.

The number of prepared objects in a single program is limited by the available memory. This includes both statement identifiers named in PREPARE statements and cursor declarations that incorporate SELECT or INSERT statements. To avoid exceeding the limit, use a FREE statement to release some statements or cursors.

The term “statement identifier” means *statement id* or *statement id variable*.

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 is converted to an internal form. This translated statement is saved for later execution in a data structure that the PREPARE statement allocates. The structure has the name *statement identifier*. Subsequent SQL statements refer to the statement using the *statement identifier*.

A subsequent FREE statement releases the resources allocated to the statement. After you release the database server resources, you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.

A program can consist of one or more source code files. By default, the scope of a statement identifier is global to the program. This means that a statement identifier 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, you should preprocess all the files with the **-local** command line option. See your ESQL product manual for more information, restrictions, and performance issues when preprocessing with the **-local** option.

Releasing a Statement Identifier

A statement identifier can represent only one SQL statement or sequence of statements at a time. You can execute a new PREPARE statement with an existing statement identifier, if you wish to bind a given statement identifier to different SQL statement text.

The PREPARE statement supports dynamic statement identifier names, allowing you to prepare a statement identifier as an identifier or as a host character string variable. In the following pairs of examples, the first example shows a statement identifier prepared as an embedded-language variable and the second shows it as a character string constant.

Figure 7-61

Preparing a statement identifier in IBM Informix ESQL/C

```
strcpy (stmtid, "query2");
$ PREPARE $stmtid FROM
    "SELECT * FROM customer";

$ PREPARE query2 FROM
    "SELECT * FROM customer";
```

Figure 7-62

Preparing a statement identifier in IBM Informix ESQL/COBOL

```
MOVE "QUERY_2" TO STMTID.
EXEC SQL
    PREPARE :STMTID FROM
        "SELECT * FROM CUSTOMER"
END-EXEC.

EXEC SQL
    PREPARE QUERY_2 FROM
        "SELECT * FROM CUSTOMER"
END-EXEC.
```

A *statement id variable* must be of the CHARACTER data type. In C, it must be defined as `$char`. In COBOL, *id variables* must be declared as a standard CHARACTER type. ♦

Statement Text

The PREPARE statement can take statement text either as a quoted string or as text 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.
Comments preceded by "--" (two hyphens), or enclosed in "{" (curly braces) are standard in SQL and are allowed in the statement text. The comment ends at the end of the line or at the end of the statement.
- The text can contain either a single SQL statement or a sequence of statements separated by semicolons.
- 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 names of tables and columns. Therefore, you cannot prepare a SELECT statement that contains an INTO clause, since the INTO clause requires a host-language variable. Use a ? as a placeholder to indicate where data should be supplied when the statement is executed.
- The text cannot include an embedded SQL statement prefix or terminator, such as the dollar sign, semicolon, or the words EXEC SQL. ♦

Here is an example of a PREPARE statement in IBM Informix 4GL.

Figure 7-63

Sample PREPARE statement in IBM Informix 4GL

```
PREPARE new_cust FROM
  "INSERT INTO customer (fname, lname) ",
  "VALUES (?, ?) "
```

Permitted Statements

You can prepare any single SQL statement except the ones in the following list:

CLOSE	EXECUTE IMMEDIATE	FREE	PUT
DECLARE	FETCH	OPEN	WHENEVER
EXECUTE	FLUSH	PREPARE	

In addition to the preceding statements, you also cannot prepare the LOAD and UNLOAD statements. ♦

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. You should use DECLARE, OPEN, and FETCH cursor statements to retrieve the rows.

A prepared SELECT statement can include a FOR UPDATE clause. This clause normally is used with the DECLARE statement to create an update cursor. An example segment of 4GL code follows.

Figure 7-64

Preparing a SELECT statement with a FOR UPDATE clause in IBM Informix 4GL

```
PREPARE up_sel FROM
  "SELECT * FROM customer ",
  "WHERE customer_num between ? and ? ",
  "FOR UPDATE"

DECLARE up_curs CURSOR FOR up_sel

OPEN sel_cursor USING low_cust, high_cust
```

Multistatement Prepares

You cannot use the following statements (in addition to the ones listed previously) in a text that contains multiple statements separated by semicolons:

CLOSE DATABASE	DATABASE	SELECT
CREATE DATABASE	DROP DATABASE	START DATABASE

Thus, a SELECT statement is not allowed in a multistatement prepare; the statements that could cause the current database to be closed in the middle of executing the sequence of statements also are not allowed.

Preparing Statements When Parameters Are Known

In some prepared statements, all needed information is known at the time the statement is prepared. Here is an example in IBM Informix ESQL/C in which two statements are prepared from constant data.

Figure 7-65

Preparing two statements from constant data in IBM Informix ESQL/C

```
sprintf(redo_st, "%s; %s",
        "DROP TABLE workt1",
        "CREATE TABLE workt1 (wtk serial, wtv float)" );
$PREPARE redotab FROM redo_st;
```

Although all parts of the statement are known prior to the prepare, they also can be derived dynamically from program input. In this IBM Informix 4GL example, user input is incorporated into a SELECT statement, which is then prepared and associated with a cursor.

Figure 7-66

Preparing 4GL statement text, including user input

```
DEFINE u_po LIKE orders.po_num
PROMPT "Enter p.o. number please: " FOR u_po
PREPARE sel_po FROM
    "SELECT * FROM orders ",
    "WHERE po_num = '", u_po, "'"
DECLARE get_po CURSOR FOR sel_po
```

For further information, consult the manual for your application development tool.

Preparing Statements That Receive Parameters Later

In some statements, parameters are not known 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 ? placeholder where a parameter must be supplied when the statement is executed.

The PREPARE statements in the following examples show some of the uses of ? placeholders.

Figure 7-67

Preparing 4GL statements with ? placeholders

```
PREPARE s3 FROM
  "SELECT * FROM customer WHERE state MATCHES ?"

PREPARE in1 FROM
  "INSERT INTO manufact VALUES (?,?,?)"

PREPARE update2 FROM
  "UPDATE customer SET zipcode = ?"
  "WHERE CURRENT OF zip_cursor"
```

You only can use a placeholder to supply a value for an expression. You cannot use a ? placeholder to represent an identifier such as a database name, a table name, or a column name.

The following example segment of IBM Informix ESQL/C code prepares a statement from a variable named **demoquery**. The text in the variable includes one ? 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.

Figure 7-68

Preparing an IBM Informix ESQL/C statement that receives values

```
$char queryvalue [6];
$char demoquery [80];
$database stores5;
sprintf(demoquery, "%s %s",
  "SELECT fname, lname FROM customer",
  "WHERE lname > ? ");
$PREPARE quid FROM $demoquery;
$DECLARE democursor CURSOR FOR quid;
strcpy(queryvalue, "C");
$OPEN democursor USING $queryvalue;
```

The USING clause is available in both OPEN (for statements associated with a cursor) and EXECUTE (all other prepared statements) statements. An IBM Informix 4GL example follows.

Figure 7-69

Executing a 4GL prepared SELECT statement using an OPEN statement

```
DEFINE zip LIKE customer.zipcode
PREPARE zip_sel FROM
  "SELECT * FROM customer WHERE zipcode MATCHES ?"
DECLARE zip_curs CURSOR FOR zip_sel
PROMPT "Enter a zipcode: " FOR zip
OPEN zip_curs USING zip
```

If the prepared SELECT statement contains a ? placeholder, you cannot execute the statement with a FOREACH statement; you must use the OPEN, FETCH, and CLOSE group of statements. ♦

Preparing Statements with SQL Identifiers

You cannot use ? placeholders for SQL identifiers such as a database name, a table name, or a column name; you must specify these identifiers in the statement text when it is prepared.

However, if these identifiers are not available when the statement is written, you can construct a statement that receives SQL identifiers from user input. In the following 4GL example, the name of a column is supplied by the user and inserted in the statement text before the PREPARE statement. The search value in that column also is taken from user input, but it is supplied to the statement with a USING clause.

Figure 7-70

Preparing an 4GL statement that receives SQL identifiers as input

```

DEFINEcolumn_name CHAR(30),
      column_value CHAR(40),
      del_str CHAR(100)

PROMPT "Enter column name: " FOR column_name

LET del_str =
  "DELETE FROM customer WHERE ",
  column_name CLIPPED, "=? "
PREPARE de4 FROM del_str

PROMPT "Enter search value in column ",column_name, ":"
      FOR column_value

EXECUTE de4 USING column_value

```

The IBM Informix ESQL/C program fragment in the next example prompts the user for the name of a table and uses that name in a SELECT statement. Because the table name is not known until run time, the number and data types of the table columns also are 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 using the descriptor. The fetch puts each row into memory locations dynamically provided by the program.

If a program were to retrieve all rows in the active set, the FETCH statement would be placed in a loop that fetched each row. If the FETCH statement retrieved more than one data value (column), there would be another loop after the FETCH, which performed some action on each data value.

Figure 7-71

Preparing an IBM Informix ESQL/C statement that receives SQL identifiers as input

```
#include <stdio.h>
#include sqlca;
#include sqllda;
#include sqltypes;

char *malloc( );

main()
{
    struct sqllda *demodesc;
    $char demoselect[200];
    char tablename[19];
    int i;

    /* This program selects all the columns of a given tablename.
       The tablename is supplied interactively. */

    $database stores5;

    printf( "This program does a SELECT * on a table\n" );
    printf( "Enter table name: " );
    scanf( "%s",tablename );

    sprintf( demoselect, "select * from %s", tablename );

    $prepare iid from $demoselect;
    $describe iid into demodesc;

    /* Print what DESCRIBE returns */

    for ( i = 0; i < demodesc->sqlld; i++ )
        prsqllda (demodesc->sqllvar + i);

    /* Assign the data pointers. */

    for ( i = 0; i < demodesc->sqlld; i++ ) {
        switch (demodesc->sqllvar[i].sqltype & SQLTYPE) {
            case SQLCHAR:
                demodesc->sqllvar[i].sqltype = CCHARTYPE;
                demodesc->sqllvar[i].sqlllen++;
                demodesc->sqllvar[i].sqldata =
                    malloc( demodesc->sqllvar[i].sqlllen );
                break;

            case SQLSMINT:
            case SQLINT:
            case SQLSERIAL:
                demodesc->sqllvar[i].sqltype = CINTTYPE;
                demodesc->sqllvar[i].sqldata =
                    malloc( sizeof( int ) );
                break;
        }
    }
}
```



```

        /* And so on for each type. */
    }
}

/* Declare and open cursor for select . */
$prepare d_stmt from $demoselect;
$declare d_curs cursor for d_stmt;
$open d_curs;

/* Fetch selected rows one at a time into demodesc. */
for( ; ; ) {
    printf( "\n" );
    $fetch d_curs using descriptor demodesc;
    if ( sqlca.sqlcode != 0 )
        break;
    for ( i = 0; i < demodesc->sqlld; 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... */
        }
    }
}
$close d_curs;
$free d_curs;

/* Free the data memory. */
for ( i = 0; i < demodesc->sqlld; i++ )
    free( demodesc->sqlvar[i].sqldata );

printf ("Program Over.\n");
}

prsqlda(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=%lx\n", sp->sqlname);
    }

```

Preparing Sequences of Multiple SQL Statements

You can execute several SQL statements as one action if you include them all 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 action that occurs in a previous statement in the text.

All compiled products return error status information on the first error in the multistatement text. There is no indication of which statement in the sequence caused an error.

The following example contains a fragment of IBM Informix 4GL code that updates the **stores5** database by replacing the existing manufacturer codes with new codes. Since the **manu_code** columns are potential join columns that link four of the tables, the new codes must replace the old codes in three tables.

Figure 7-72

Preparing multistatement text in IBM Informix 4GL

```
DATABASE stores5
MAIN
  DEFINE code_chnge RECORD
    new_code LIKE manufact.manu_code,
    old_code LIKE manufact.manu_code
  END RECORD,
  sqlmulti CHAR(250)

  PROMPT "Enter new manufacturer code: "
  FOR code_chnge.new_code
  PROMPT "Enter old manufacturer code: "
  FOR code_chnge.old_code
  LET sqlmulti =
    "UPDATE manufact SET manu_code = ? WHERE manu_code = ?;",
    "UPDATE stock SET manu_code = ? WHERE manu_code = ?;",
    "UPDATE items SET manu_code = ? WHERE manu_code = ?;",
    "UPDATE catalog SET manu_code = ? WHERE manu_code = ?;"

  PREPARE exmulti FROM sqlmulti
  EXECUTE exmulti USING code_chnge.*, code_chnge.*, code_chnge.*
    code_chnge.*
  END MAIN
```

In the next example, six SQL statements are prepared into a single IBM Informix ESQL/C string query. Individual statements are delimited with semicolons. A single **\$prepare** statement can prepare all six statements for execution and a single **\$execute** statement can execute the **qid** string.

Figure 7-73*Preparing multistatement text in IBM Informix ESQL/C*

```
sprintf (query, "%s %s %s %s %s %s %s %s %s",
        "begin work;",
        "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 (?, ?);",
        "commit work;");
$prepare qid from $query;
$execute qid using
        $delta, $acct_number, $delta, $teller_number,
        $delta, $branch_number, $timestamp,$values;
```

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 caused by redundant parsing and optimizing. For example, an UPDATE statement 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. An 4GL example follows.

Figure 7-74*Preparing an 4GL statement to improve performance*

```
PREPARE up1 FROM "UPDATE customer ",
        "SET discount = 0.1 WHERE customer_num = ?"
WHILE TRUE
        PROMPT "Enter Customer Number" FOR dis_cust
        IF dis_cust = 0 THEN
                EXIT WHILE
        END IF
        EXECUTE up1 USING dis_cust
END WHILE
```

References

In this manual, see the following statements: DECLARE, DESCRIBE, EXECUTE, FREE, and OPEN.

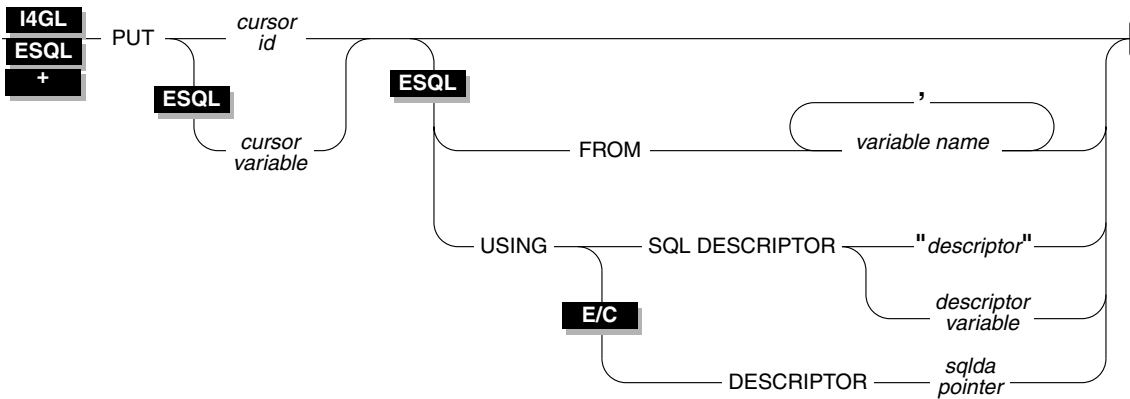
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of PREPARE statements and dynamic SQL.

PUT

Purpose

Use the PUT statement to store a row in an insert buffer for later insertion into the database.

Syntax



- cursor id* is the identifier of a cursor declared for an INSERT statement.
- cursor variable* is an embedded variable name that identifies a cursor declared for an INSERT statement.
- descriptor* is a quoted string that identifies a system descriptor area allocated with the ALLOCATE DESCRIPTOR statement.
- descriptor variable* is an embedded variable name that identifies a system descriptor area allocated with the ALLOCATE DESCRIPTOR statement.
- sqlda pointer* points to an **sqlda** structure representing values that correspond to the ? placeholders in a prepared INSERT statement.
- variable name* is a program variable whose contents are to replace a ? placeholder in a prepared INSERT statement.

Usage

Each PUT statement stores a row in an insert buffer that was created when *cursor name* was opened. If the buffer has no room for the new row when the statement is executed, the buffered rows are written to the database in a block and the buffer is emptied. As a result, some PUT statement executions cause database output and some do not.

You can use the FLUSH statement to write buffered rows to the database without adding a new row. The CLOSE statement writes any remaining rows before it closes an insert cursor.

If the current database uses explicit transactions, you must execute a PUT statement within a transaction.

Here is an 4GL example of the use of a PUT statement.

Figure 7-75

Using the PUT statement in IBM Informix 4GL

```
DECLARE ins_curs CURSOR FOR
  INSERT INTO state VALUES (code, sname)
OPEN ins_curs
LET code = "AK"
LET sname = "Alaska"
PUT ins_curs
```

Here is an example using IBM Informix ESQL/C and a prepared statement.

Figure 7-76

Using the PUT statement in IBM Informix ESQL/C

```
$PREPARE ins_mcode FROM "INSERT INTO manufact VALUES(?,?)";
$DECLARE mcode CURSOR FOR ins_mcode;
$OPEN mcode;
$PUT mcode FROM $the_code, $the_name;
```

X/O

PUT is not an X/Open standard SQL statement. Therefore, you will see a warning message if you compile a PUT statement in X/Open mode in an ESQL product. For details on compiling in X/Open mode, see your product manual. ♦

Supplying Inserted Values

The values that compose the inserted row can come from one of four sources:

- Constant values written into the INSERT statement
- Program variables named in the INSERT statement
- Program variables named in the FROM clause of the PUT statement
- Values that are prepared in memory addressed by an **sqllda** 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* of 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 IBM Informix 4GL example, 99 empty customer records are inserted into the **customer** table. Since all values are constants, no disk output occurs until the cursor is closed. (The constant zero for **customer_num** causes generation of a SERIAL value.)

Figure 7-77

Inserting empty customer records into a table in IBM Informix 4GL

```
DECLARE fill_c CURSOR FOR
  INSERT INTO customer(customer_num) VALUES(0)
DEFINE count SMALLINT
OPEN fill_c
FOR count = 1 TO 99
  PUT fill_c
END FOR
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 compose the row that is inserted into the buffer.



Tip: You can only name program variables in the VALUES clause when the INSERT statement is written as part of the DECLARE statement. Variable names are not recognized in the context of a prepared statement that is associated with a cursor through its statement identifier.

The IBM Informix ESQL/C example that follows 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 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 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.

Figure 7-78

Using an insert cursor in IBM Informix ESQL/C

```
int keep_going = 1;
$struct cust_row { /* fields of a row of customer table */ } cust_rec;
$declare ins_curs cursor for
    insert into customer values ($cust_rec);
$open ins_curs;
for (; (sqlca.sqlcode == 0) && (keep_going) ;)
{
    keep_going = get_user_input(cust_rec); /* ask user for new customer */
    if (keep_going)/* user did supply customer info */
    {
        cust_rec.customer_num = 0; /* request new serial value */
        $put ins_curs;
    }
    if (sqlca.sqlcode == 0)/* no error from PUT */
        keep_going = (prompt_for_y_or_n("another new customer") == 'Y')
}
$close ins_curs;
```

Naming Program Variables in PUT

When the INSERT statement is prepared (see the PREPARE statement on page 7-218), you cannot use program variables in its VALUES clause. However, you can represent values using a ? placeholder. You supply the missing values by listing the names of program variables in the FROM clause of the PUT statement. An example in IBM Informix 4GL follows.

Figure 7-79

Listing program variables in a PUT statement in IBM Informix 4GL

```
DEFINE answer CHAR(1), u_company LIKE customer.company
PREPARE sel2 FROM
    "INSERT INTO customer (customer_num, company) ",
    "VALUES (0, ?)"

DECLARE ins_curs CURSOR FOR sel2
OPEN ins_curs

LET answer = "y"
WHILE answer = "y"
    PROMPT "Enter a customer: " FOR u_company
    PUT ins_curs FROM u_company
    PROMPT "Do you want to enter another customer (y/n) ? "
    FOR answer
END WHILE

CLOSE ins_curs
```

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 then can specify that system descriptor area in the USING SQL DESCRIPTOR clause of the PUT statement.

For details on using descriptors, see [Chapter 6](#) in this manual and the manual for the embedded-language product you are using. The following examples show how to associate values from a system descriptor area.

Figure 7-80

Sample PUT USING SQL DESCRIPTOR statement in IBM Informix ESQL/C

```
$ PUT selcurs USING SQL DESCRIPTOR "desc1";
```

ESQL

Figure 7-81

Sample PUT USING SQL DESCRIPTOR statement in IBM Informix ESQL/COBOL

```
EXEC SQL PUT SEL_CURS USING SQL DESCRIPTOR "DESC1" END-EXEC.
```



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 is executed, the values described by the **sqlda** are used to replace ? 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.

For details on the **sqlda** structure, see [Chapter 6](#) in this manual and the *IBM Informix ESQL/C Programmer's Manual*.

Figure 7-82

Sample PUT USING DESCRIPTOR statement in ESQL/C

```
$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; an activity 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 is executed.
- A CLOSE statement closes the cursor.
- An OPEN statement is executed naming the cursor.

When applied to an open cursor, the OPEN statement closes the cursor before reopening it; this implied CLOSE statement flushes the buffer.

- A COMMIT WORK statement is executed.

If the program terminates without closing an insert cursor, the buffer remains unflushed. Rows 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 fields of the SQLCA, as shown in the following table.

4GL	ESQL/C	ESQL/COBOL
STATUS SQLCA.SQLCODE	sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA
SQLCA.SQLERRD[3]	sqlca.sqlerrd[2]	SQLERRD[3] OF SQLCA

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 only discovered when the buffer is flushed. When an error is discovered, rows in the buffer located after the error are *not* inserted; they are lost from memory.

The SQLCODE field is set to zero 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 both are set to zero (SQLCODE because there was no error 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 zero and SQLERRD is set to the number of rows successfully inserted into the database.
- If an error occurs while the buffered rows are written to the database, SQLCODE indicates the error, while SQLERRD contains the number of rows that were successfully inserted. (The uninserted rows are discarded from the buffer.)

Counting Total and Pending Rows

To count the number of rows actually inserted in the database and the number not yet inserted, follow this procedure:

- Prepare two integer variables, for example, **total** and **pending**.
- When the cursor is opened, set both variables to zero.
- Each time a PUT statement is executed, increment both **total** and **pending**.
- Whenever a PUT or FLUSH statement is executed, or the cursor is closed, subtract the third field of the SQLERRD array from **pending**.

At any time, **total** minus **pending** is the number of rows actually inserted. If all commands are successful, **pending** should contain zero after the cursor is closed. If an error occurs during a PUT, FLUSH, or CLOSE statement, the value remaining in **pending** is the number of uninserted (discarded) rows.

References

In this manual, see the following cursor-related statements: CLOSE, FLUSH, DECLARE, and OPEN. See also the ALLOCATE DESCRIPTOR statement.

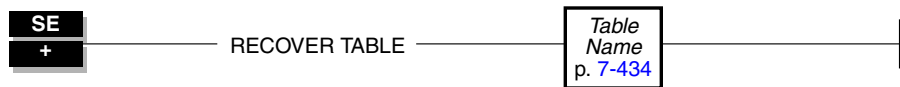
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of the PUT statement.

RECOVER TABLE

Purpose

Use the RECOVER TABLE statement with IBM Informix SE to restore a database table in the event of failure.

Syntax



Usage

The RECOVER TABLE statement applies the table audit trail to an archive copy of the database. IBM Informix SE uses audit trails to record operations on a per-table basis. You can issue a RECOVER TABLE statement if you own the table or have DBA privilege on the database.

In the event of a system failure, use an operating system utility to restore each of the table files for which you have an audit trail. Issue the RECOVER TABLE statement to update each newly restored table with the transactions recorded in the audit trail.

Backup/Restore Procedure

The recommended backup/restore procedure for making archive copies of a database that includes audit trails is as follows:

- Execute the DROP AUDIT statement for each table that has an audit trail. The DROP AUDIT statement ends system logging to the audit trail files.
- Execute the CREATE AUDIT statement for each table, specifying the pathname of the new audit trail. For maximum protection, specify a location that is not on the same storage device as the database. You also can select a filename that reflects the table name and the sequence of the file in the audit trail; for example, **audit_cust_001** or **audit_cust_002**. The CREATE AUDIT statement registers the new name and location of the audit trail file in the **sysables** system catalog table.
- Back up the database files using an operating system utility.

During execution, the RECOVER TABLE statement checks that the audit trail and *table name* have consistent record numbers for rows where changes occurred. In extremely rare instances, the RECOVER TABLE statement can find an inconsistency caused by a system crash. In this case only, the RECOVER TABLE statement stops and you must restore the table manually.

The following list of actions and statements serves as a guide for a recovery of the **customer** table. First, restore the **customer** table from your last archive copy. Second, run the following statements, which assume that your audit trail began immediately after you created the archive copy:

```
RECOVER TABLE customer
DROP AUDIT FOR customer
CREATE AUDIT FOR customer
```

Third, create a new backup of the recovered table.

The audit trail file is not in human-readable form. Even so, it is possible for the DBA to copy the file to a database (**.dat**) file and manipulate the file, if desired. The modified file can be copied back to the audit trail file, enabling customized restorations of particular tables. For example, you can modify the audit trail file to exclude rows entered by a particular user or to undo specific transactions. For specific instructions on modifying audit trail files, refer to the manual for your application development tool.

References

In this manual, see the following statements: CREATE AUDIT and DROP AUDIT.

RENAME COLUMN

Purpose

Use the RENAME COLUMN statement to change the name of a column.

Syntax

```

+ RENAME COLUMN Table  
Name  
p. 7-434 .old column TO new column
  
```

new column is the new name of the column.

old column is the column you are renaming.

Usage

You can rename a column of a table if any one of the following conditions is true:

- You own the table.
- You have DBA privilege on the database.
- You have Alter privilege on the table.

When you rename a column, choose a column name that is unique within the table.

If the column is referenced by a view in the database, the text of the view in the **sysviews** system catalog table is updated to reflect the new column name. If the column is referenced by a check constraint in the database, the text of the check constraint in the **syschecks** system catalog table is updated to reflect the new column name. ♦

DB

ESQL

The following example assigns the **customer_num** column in the **customer** table the new name of **c_num**:

```
RENAME COLUMN customer.customer_num TO c_num
```

You cannot use a ROLLBACK WORK statement to undo a RENAME COLUMN statement that successfully executes. If you roll back a transaction that contains a RENAME COLUMN statement, the column retains its new name and you do not receive an error message. ♦

References

In this manual, see the following statements: ALTER TABLE, CREATE TABLE, and RENAME TABLE.

RENAME TABLE

Purpose

Use the RENAME TABLE statement to change the name of a table.

Syntax

```

+ — RENAME TABLE — owner. old name — TO — new name —
  
```

new name is the new name you are assigning to the table.

old name is the table you are renaming.

owner is the owner of the table.

Usage

You can rename a table if any one of the following statements is true:

- You own the table.
- You have DBA privilege on the database.
- You have Alter privilege on the table.

You cannot change the table owner by renaming the table. You can specify *owner* as part of *old name*, but an error occurs during compilation if you try to specify *owner* as part of *new name*.

If a view references this table, the text of the view in the **sysviews** system catalog table is updated to reflect the new table name.

In an ANSI-compliant database, you must specify *owner* if you are referring to a table that you do not own. ♦

ANSI

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 four steps:

- Create a new table, **new_table**, that contains the column **quantity** in the third position.
- Fill the table with data from the current **items** table.
- Drop the old **items** table.
- Rename **new_table** with the name **items**.

Figure 7-83

Reorganizing the items table using the RENAME TABLE statement

```
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
```

SE

You cannot use a ROLLBACK WORK statement to undo a RENAME TABLE statement that successfully executes. If you roll back a transaction that contains a RENAME TABLE statement, the table retains its new name and you do not receive an error message. ♦

References

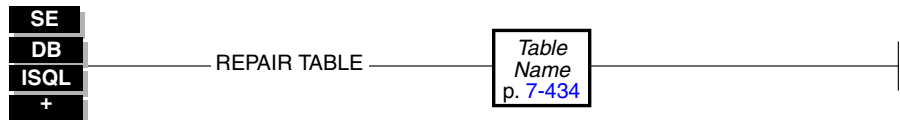
In this manual, see the following statements: ALTER TABLE, CREATE TABLE, DROP TABLE, and RENAME COLUMN.

REPAIR TABLE

Purpose

Use the REPAIR TABLE statement to remove and rebuild table indexes or data that may have been damaged or corrupted because of a power failure, computer crash, or other unexpected program stoppage. Only those tables that are found to be damaged are rebuilt. You can determine whether you need to use the REPAIR TABLE statement by first issuing the CHECK TABLE statement.

Syntax



Usage

Specify the name of the table for which you want to restore the integrity of the index files. For example:

```
REPAIR TABLE cust_calls
```

You cannot use the REPAIR TABLE statement on a table unless you own it or have DBA privilege on the database. You cannot use the REPAIR TABLE statement on the system catalog table **systables** unless you have the DBA privilege on the database.

The REPAIR TABLE statement calls the **bcheck** utility. See the *IBM Informix SE Administrator's Guide* for a full description of **bcheck**.

References

In this manual, see the CHECK TABLE statement.

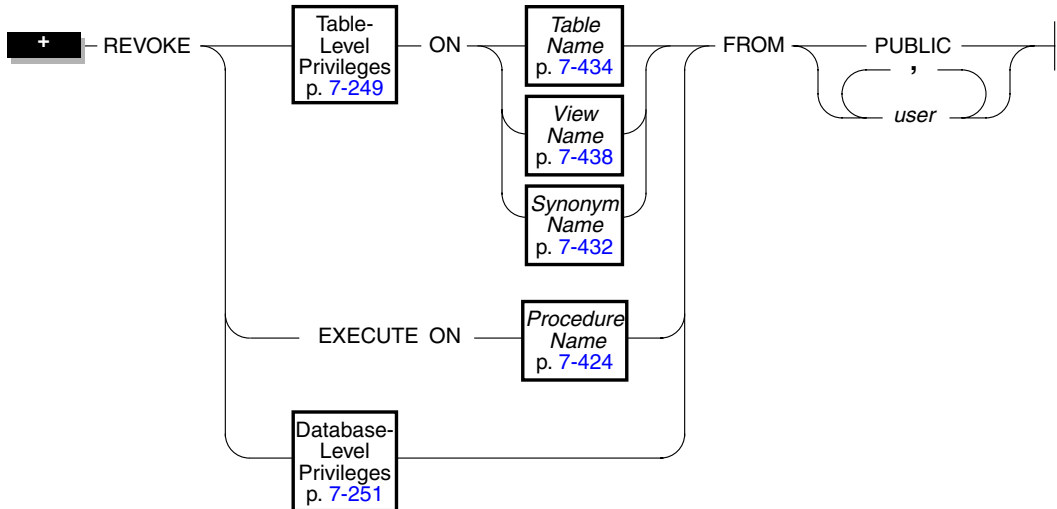
In the *IBM Informix SE Administrator's Guide*, for a discussion of the **bcheck** utility.

REVOKE

Purpose

Use the REVOKE statement to take away another user's privileges for a table, database, or procedure.

Syntax



user names the user or users whose privileges are revoked. The keyword PUBLIC revokes privileges from all users.

Usage

You can use the REVOKE statement with the GRANT statement to finely control the ability of users to modify the database and access and modify data in the tables.

You can revoke all or some of the privileges that you granted to other users. No one can revoke privileges that another user granted. However, 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.

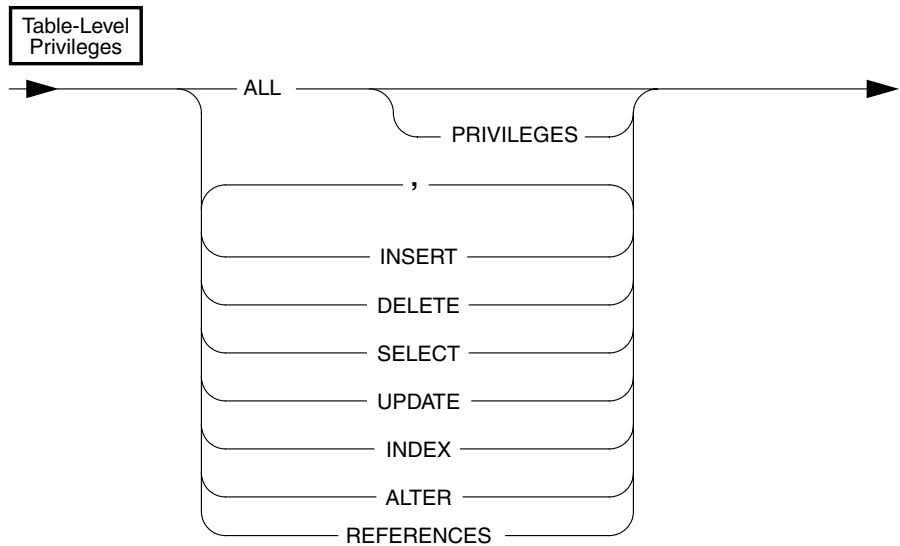
If you revoke the EXECUTE privilege on a stored procedure from a user, the user can no longer run that procedure using either the EXECUTE PROCEDURE or CALL statements.

Users cannot revoke privileges from themselves.

SE

You cannot use a ROLLBACK WORK statement to undo a REVOKE statement that successfully executes. If you roll back a transaction that contains a REVOKE statement, the privilege is not regranted to the user and you do not receive an error message. ♦

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, then both of them must revoke the privilege. If only one of the grantors 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 Select privilege to PUBLIC and then attempts to revoke 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 that would represent the grant that the table owner is now trying 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. Note that, unlike the GRANT statement, you cannot qualify the Select, Update, or References privilege with a column name in a REVOKE statement. That is, you cannot revoke access on specific columns.

SELECT is the ability to display data obtained from a SELECT statement.

UPDATE is the ability to change column values.

INSERT is the ability to insert rows.

DELETE is the ability to delete rows.

INDEX is the ability to create permanent indexes. You must have Resource privilege to take advantage of Index privilege. (Any user with Connect privilege can create indexes on temporary tables.)

ALTER is the ability to add or delete columns, or to modify column data types. You must have Resource privilege to take advantage of Alter privilege.

REFERENCES is the ability to reference columns in referential constraints. You must have 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 Resource privilege on the database.) You can restrict the References privilege to one or more columns by listing them.

ALL is all of the preceding privileges. The PRIVILEGES keyword is optional.

The following example revokes Index and Alter privileges from all users for the **customer** table. These privileges are then granted specifically to user **mary**.

Figure 7-84

Revoking and granting Index and Alter privileges

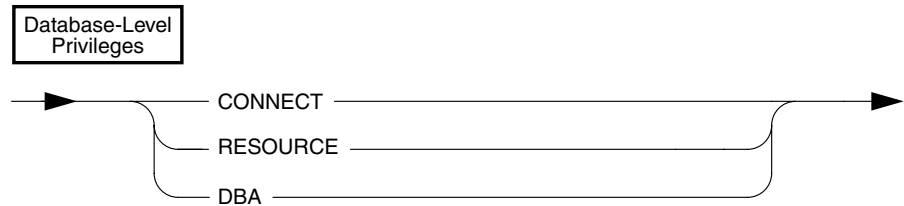
```
REVOKE INDEX, ALTER ON customer FROM PUBLIC
GRANT INDEX, ALTER ON customer TO mary
```


Since you cannot revoke access on specific columns, when you revoke Select, Update, or References privilege from a user, you revoke the privilege for all columns in the table. You must use a GRANT statement to specifically regrant any column-specific privilege that should be available to the user.

Figure 7-85
Regranting column-specific privileges

```
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 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 follow), if you revoke either Resource or Connect privilege from a user with DBA privilege, the statement has no effect. If you revoke DBA privilege from a user with DBA privilege, the user retains Connect privilege on the database. To deny database access to a user with DBA or Resource privilege, you must first revoke the DBA or Resource privilege, and then revoke the Connect privilege in a separate REVOKE statement.

Similarly, if you revoke Connect privilege from a user with Resource privilege, the statement has no effect. If you revoke Resource privilege from a user with Resource privilege, the user retains Connect privilege on the database.

The three database privileges are associated with the following keywords:

- CONNECT** Connect privilege gives you the ability to query and modify data. You can modify the database schema if you own the object you wish to modify. Any user with Connect privilege can perform the following functions:
- Execute SELECT, INSERT, UPDATE, and DELETE statements, provided the user has the necessary table-level privileges.
 - Create views, provided the user has 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, provided the user owns the table (or has been given privileges on the table with the WITH GRANT OPTION keywords).
- RESOURCE** Resource privilege 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

SE



DBA

In addition to the capabilities of the Resource privilege, the holder of DBA privilege can perform the following functions:

- Grant any privilege, including DBA privilege, to another user.
- Use the NEXT SIZE keywords to alter extent sizes in the system catalog tables
- Insert, delete, or update rows of any system catalog table except **sys**tables.
- Drop any object, regardless of who owns it.
- Create tables, views, and indexes, and specify another user as owner of the objects.
- Execute the DROP DATABASE command.
- Execute the START DATABASE, and ROLLFORWARD DATABASE commands. ♦

Tip: Although user *informix* can modify the system catalog tables, it is strongly recommended that you do not update, delete, or alter any rows in these tables. Modifying the system catalog tables can destroy the integrity of the database.

References

In this manual, see the following statement: GRANT.

In *IBM Informix Guide to SQL: Tutorial*, see the discussion of privileges and security.

ROLLBACK WORK

Purpose

Use the ROLLBACK WORK statement to cancel a transaction and undo any changes that occurred since the beginning of the transaction.

Syntax

ROLLBACK WORK

Usage

The ROLLBACK WORK statement is valid only in databases with transactions.

In a database that is not ANSI-compliant, you 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.

The ROLLBACK WORK statement releases all row and table locks held by the cancelled transaction. 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. If you issue a ROLLBACK WORK statement when no transaction is pending, the statement is accepted but has no effect. ♦

ANSI

SE

If you are using IBM Informix SE, a ROLLBACK WORK statement undoes all database changes except those that result from GRANT or REVOKE statements or from data definition statements. Data definition statements are treated as single transactions. If they were executed successfully, they are committed automatically and cannot be rolled back by the ROLLBACK WORK statement. Data definition statements include statements that modify the number, names, or indexes of tables and statements that modify the number, names, or data types of columns.

If a transaction is rolled back, the actions taken to undo the transaction are also logged to table audit trails, if any exist. ♦

I4GL

The ROLLBACK WORK statement closes all open cursors except those declared with hold, which remain open despite transaction activity. ♦

ESQL

I4GL

Do not use a ROLLBACK WORK statement within a FOREACH loop. ♦

ESQL

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

References

In this manual, see the following statements: BEGIN WORK and COMMIT WORK.

If you are using 4GL, refer to the FOREACH and WHENEVER statements in the manual for your application development tool.

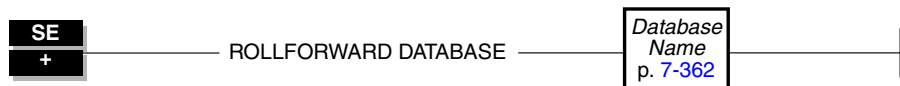
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of ROLLBACK WORK.

ROLLFORWARD DATABASE

Purpose

Use the ROLLFORWARD DATABASE statement with the IBM Informix SE database server to apply the transaction log file to a restored database.

Syntax



Usage

To restore a database, you need both the archive copy of the database and the transaction log that was begun immediately after the archive copy was made.

To execute the ROLLFORWARD DATABASE statement, you need DBA privilege. Always precede a ROLLFORWARD DATABASE statement with a CLOSE DATABASE statement. The ROLLFORWARD DATABASE statement fails if a database is open.

The ROLLFORWARD DATABASE statement sets an exclusive lock on the database to prevent access by other processes. If another process is using the database (even if the database is only being read), the ROLLFORWARD DATABASE statement fails.

The database remains locked after the ROLLFORWARD DATABASE statement executes. This allows you to check for errors before you give other users access. When you are satisfied that the database is ready for use, release the exclusive lock by executing the CLOSE DATABASE statement. You can open the database with the DATABASE statement.

If you are using IBM Informix NET, you must be working on a database server to issue a ROLLFORWARD DATABASE statement. You cannot execute the statement from a client. ♦

INET

References

In this manual, see the following statements: BEGIN WORK, COMMIT WORK, CLOSE DATABASE, DATABASE, and ROLLBACK WORK.

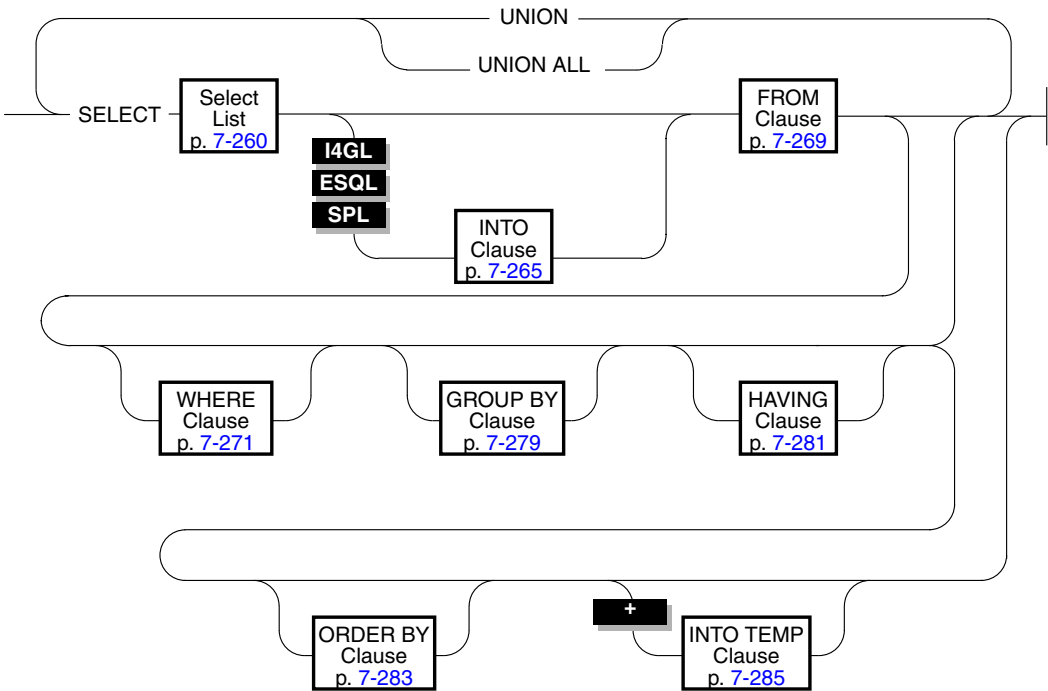
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of archives and logs.

SELECT

Purpose

Use the SELECT statement to query a database.

Syntax



Usage

You can query the tables in the current database or in a database that is not current.

STAR

You can query the tables in the databases that are on a different database server from your current database. ♦

SE

You only can query the current database. ♦

The SELECT statement is made up of eight basic parts. Each part is listed here and explained in detail on the following pages:

SELECT clause names a list of items to be read from the database.

INTO clause specifies the program variables or host variables that receive the selected data. ♦

I4GL**ESQL****SPL**

FROM clause names the tables that contain the selected columns.

WHERE clause sets conditions on the rows that are chosen.

GROUP BY clause combines groups of rows into summary results.

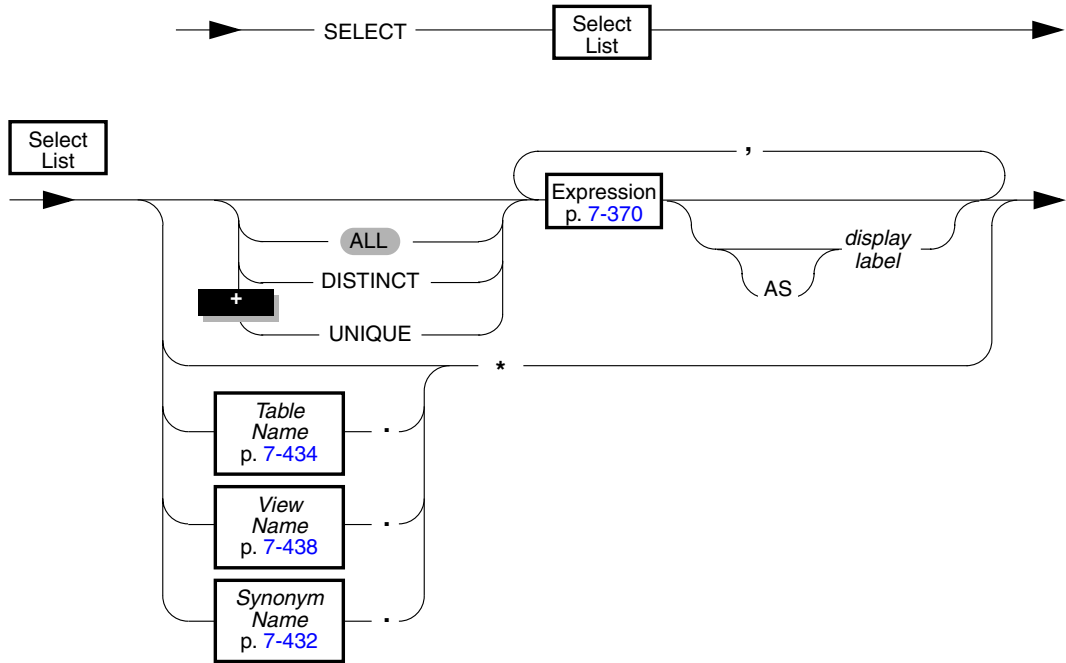
HAVING clause sets conditions on the summary results.

ORDER BY clause orders the selected rows.

INTO TEMP clause creates a temporary table in the current database and puts the results of the query into the table.

SELECT Clause

The SELECT clause contains the keyword SELECT and the list of objects to be selected, as shown in the following diagram:



display label is a temporary name that you assign to the expression.

In the SELECT clause, you specify exactly what data is being selected, as well as whether you want to omit duplicate values.

Allowing Duplicates

You can apply the keywords `ALL`, `UNIQUE`, or `DISTINCT` to indicate whether duplicate values are returned, if any exist. If you do not specify any of these keywords, all of the rows are returned by default.

<code>ALL</code>	specifies that all selected values are returned, regardless of whether there are duplicates. <code>ALL</code> is the default state.
<code>DISTINCT</code>	eliminates duplicate rows from the query results.
<code>UNIQUE</code>	eliminates duplicate rows from the query results. <code>UNIQUE</code> is a synonym for <code>DISTINCT</code> .

For example, the following query lists the `stock_num` and `manu_code` of all the 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 of the four basic types of expressions (column, constant, function, or aggregate function), or combinations thereof, in the select list. The four expression types are described in detail beginning with the section [“Expression” on page 7-370](#).

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.

Selecting Columns

Column expressions are the most common expressions used in a SELECT statement. See [“Column Expressions” on page 7-373](#) for a complete description of the syntax and use of column expressions.

Examples of column expressions within a select list follow.

Figure 7-86

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 returned by the query. See [“Constant Expressions” on page 7-376](#) for a complete description of the syntax and use of constant expressions.

Examples of constant expressions within a select list follow.

Figure 7-87

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 is an expression that 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.

Some examples of function expressions within a select list follow.

Figure 7-88

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 returned by the WHERE clause of the SELECT statement. In the absence of a WHERE clause, the aggregate functions take on values that depend on all the rows formed by the FROM clause.

Some examples of aggregate functions in a select list follow.

Figure 7-89

Aggregate functions within 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 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.

Figure 7-90

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
```

DB

ISQL

ESQL

Using a Display Label

If you are creating a temporary table, you must supply a display label for any columns that are not simple column expressions. The display label is used as the name of the column in the temporary table.

A display label appears as the heading for that column in the output of the SELECT statement. ♦

The value of *display label* is stored in the **sqlname** field of the **sqllda** structure. See [Chapter 6](#) of this manual for more information on the **sqllda** structure. ♦

If you are using the SELECT statement in creating a view, do not use display labels. Specify the desired label names in the CREATE VIEW column list instead.

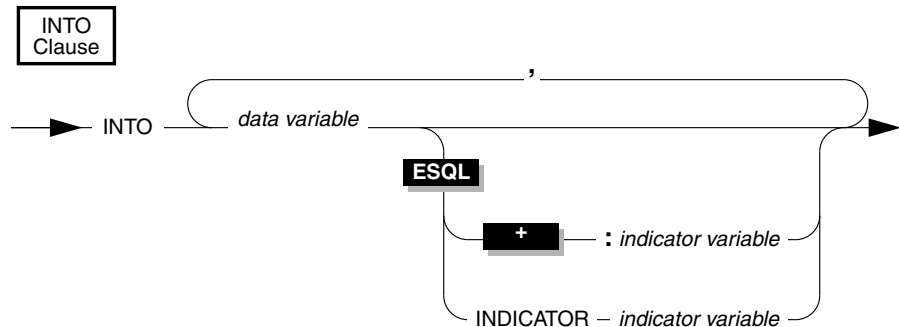
Using the AS Keyword

If your display label also is 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 to use **minute** as a display label:

```
SELECT call_dtime AS minute FROM cust_calls
```

INTO Clause

Use the INTO clause to specify the program variables or host variables to receive the data retrieved by the SELECT statement. The syntax of the INTO clause is as follows:



data variable is a program variable or host object that agrees in type and order with the corresponding columns or expressions in the select list.

indicator variable is a program variable that receives a return code if null data is placed in the corresponding *data variable*.

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 IBM Informix 4GL.

Figure 7-91

A singleton SELECT statement in IBM Informix 4GL

```
SELECT fname, lname, company_name
      INTO p_fname, p_lname, p_coname
      WHERE customer_num = 101
```

INTO Clause with Indicator Variables

You should use an indicator variable if there is the possibility that data returned from the SELECT statement is NULL. See your embedded-language product manual for more information about indicator variables.

INTO Clause with Cursors

If the SELECT statement returns more than one row, you must use a cursor to FETCH the rows one at a time. 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 IBM Informix 4GL code fragments are equivalent.

Figure 7-92

Using the INTO clause in SELECT in an IBM Informix 4GL program

```

DECLARE q_curs CURSOR FOR
  SELECT lname, company
         INTO p_lname, p_company
         FROM customer
OPEN q_curs
  WHILE status = 0
    FETCH q_curs
    ...
  END WHILE
CLOSE q_curs

```

Figure 7-93

Using the INTO clause in FETCH in an IBM Informix 4GL program

```

DECLARE q_curs CURSOR FOR
  SELECT lname, company
         FROM customer
OPEN q_curs
  WHILE status = 0
    FETCH q_curs INTO p_lname, p_company
    ...
  END WHILE
CLOSE q_curs

```

I4GL

With 4GL, you can use the FOREACH statement in place of the FETCH, OPEN, and CLOSE statements.

The following IBM Informix 4GL code fragment accomplishes the same task as the two preceding examples, except that it uses the FOREACH statement, which is available only in IBM Informix 4GL.

Figure 7-94
Using FOREACH in IBM Informix 4GL

```

DECLARE q_curs CURSOR FOR
  SELECT lname, company
  FROM customer
FOREACH q_curs INTO p_lname, p_company
  ...
END FOREACH

```

Preparing a SELECT...INTO Query

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 fetch the cursor into the program variable using the FETCH statement with an INTO clause. 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

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 used as a subscript is determined when the cursor is declared, so once declared, the subscript variable acts as a constant.

For example, the following IBM Informix 4GL code fragment declares a cursor for a SELECT...INTO statement using the variables *i* and *j* as subscripts for the array *a*. Once you declare the cursor, the INTO clause of the SELECT statement is equivalent to INTO a [*i*] , a [*j*].

Figure 7-95
Declaring a cursor for a SELECT...INTO statement in IBM Informix 4GL

```

LET i = 5
LET j = 2
DECLARE c CURSOR FOR
  SELECT order_num, po_num INTO a[i], a[j] FROM orders
  WHERE order_num =1005 AND po_num =2865

```

I4GL

You also can 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 the cursor is declared.

You also can use program variables in a FOREACH statement to specify an element of a program array in the INTO clause. The program variables are evaluated at each loop of the FOREACH statement, rather than when the cursor is declared. ♦

Error Checking

If the number of variables listed in the INTO clause differs from the number of items in the SELECT clause, a warning is returned in the SQLWARN structure; the specific structure name is shown in the following diagram. The actual number of variables transferred is the lesser of the two numbers. See [Chapter 5](#) of this manual for information about the SQLWARN structure.

4GL	ESQL/C	ESQL/COBOL
SQLCA.SQLAWARN[4]	sqlca.sqlwarn.sqlwarn3	SQLWARN3 OF SQLWARN OF SQLCA

ANSI

If the number of variables listed in the INTO clause differs from the number of items in the SELECT clause, an error is returned. ♦

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 not possible, an error occurs and a negative value is returned in the status variable. In this case, the value in the program variable is unpredictable. The specific name of the status variable for each application development tool is shown in the following diagram:

4GL	ESQL/C	ESQL/COBOL
STATUS SQLCA.SQLCODE	sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA

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:

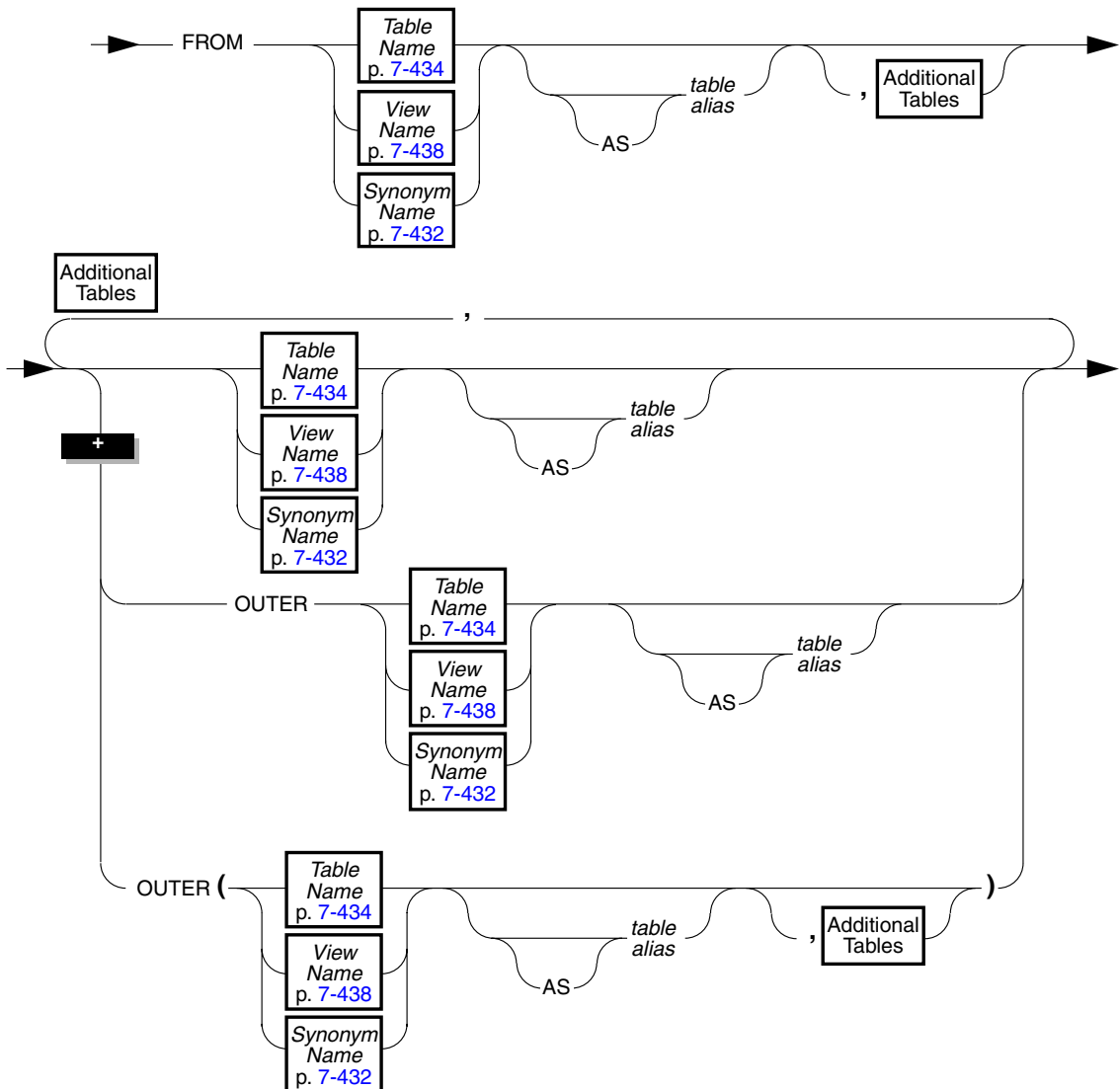


table alias is a name that you attach to the table within the scope of the SELECT statement.

Use the keyword OUTER to form outer joins. Outer joins preserve rows that otherwise would be discarded by simple joins. See the *IBM Informix Guide to SQL: Tutorial* for more information on outer joins.

You can supply an alias for a table name. You can use the alias to refer to the table in other clauses of the SELECT statement. This is especially useful with a self-join. (See the WHERE clause on page 7-271 for more information about self-joins.)

The table alias that you can specify in a FROM clause is distinct from the alias for a table that is sometimes required in the TABLES section of a form-specification file. ♦

The following examples show 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 customers who have placed orders.

Figure 7-96
Typical uses of the FROM clause

```
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 table aliases in the FROM clause and uses them in the WHERE clause.

Figure 7-97
Establishing and using table aliases

```
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.

I4GL

ISQL

Figure 7-98
Creating an outer join

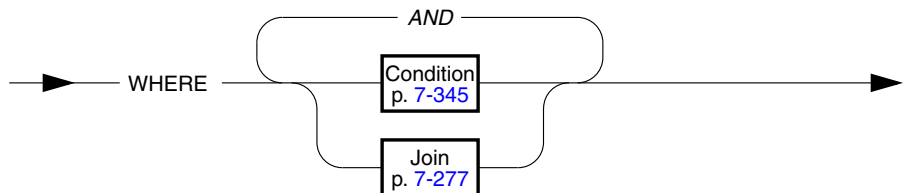
```
SELECT customer.customer_num, lname, order_num
FROM customer c, OUTER orders o
WHERE c.customer_num = o.customer_num
```

AS Keyword with Table Aliases

To use potentially ambiguous words as a table alias, you must precede them with the keyword AS. Use the AS keyword if you want to use the words ORDER, FOR, GROUP, HAVING, INTO, UNION, WHERE, WITH, CREATE, or GRANT as a table alias.

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 five 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. There are three 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 [7-345](#).

You cannot use an aggregate function in the WHERE clause unless it is part of a subquery.

Relational-Operator Condition

For a complete description of the relational-operator condition, see page [7-348](#).

A relational-operator condition is satisfied when the expressions on either side of the relational operator fulfill the relation set up by the operator. The following SELECT statements use the greater than (>) and equal (=) relational operators.

Figure 7-99
Examples of the Relational-Operator condition

```
SELECT order_num FROM orders
WHERE order_date > "6/04/91"

SELECT fname, lname, company
FROM customer
WHERE city[1,3] = "San"
```

BETWEEN Condition

For a complete description of the BETWEEN condition, see page [7-349](#).

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.

Figure 7-100
Examples of the BETWEEN condition

```
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 "6/1/90" AND "9/1/90"

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 [7-350](#).

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.

Figure 7-101
Examples of 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 [7-351](#).

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 non-null value. The following example selects the order numbers and customer numbers for which the order has not been paid.

Figure 7-102
Example of the IS NULL condition

```
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 [7-352](#).

The LIKE or MATCHES condition is satisfied when the column value meets the criteria specified in the quoted string.

The following SELECT statement returns all the columns in the **customer** table from each row in which the **lname** column begins with the literal string "Baxter". Since the string is a literal string, the condition is case sensitive.

```
SELECT * FROM customer WHERE lname LIKE "Baxter%"
```

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 "%". 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 "%".

Figure 7-103
Examples of the LIKE condition

```
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. 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 "*". The third statement uses the ESCAPE option with the MATCHES condition to retrieve rows from the **customer** table in which the **company** column includes an asterisk. The z character is used as an escape character for the wildcard "*".

Figure 7-104
Examples of the MATCHES condition

```
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 [7-350](#).

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)

```

Figure 7-105
Example of an IN subquery

EXISTS Subquery

For a complete description of the EXISTS subquery, see page [7-356](#).

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 **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)

```

Figure 7-106
Example of an EXISTS subquery

The preceding example would work equally well if you use a SELECT * in the subquery in place of the column names, since 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 [7-357](#).

In the following example, the first SELECT statement returns 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 second SELECT statement produces the same result by using the MAX aggregate function, but it may execute more quickly.

Figure 7-107

Example of an ALL subquery and an equivalent aggregate subquery

```
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 statement uses the ANY keyword; the second uses the MIN aggregate function.

Figure 7-108

Example of an ANY subquery and an equivalent aggregate subquery

```
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 will return 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.

Figure 7-109

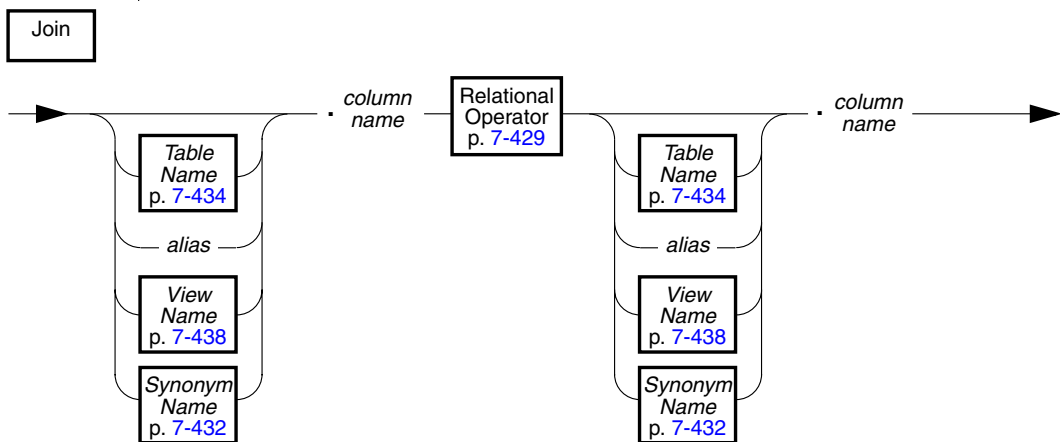
Example of a subquery omitting the ANY, ALL, or SOME keywords

```
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 in which each pair of rows (one from each table) satisfying 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:



alias is the alias assigned in the FROM clause.

column name is the name of a column in one of the tables.

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
```

Note that you do not have to select the column on which the two tables are joined.

Figure 7-110
Example of a two-table join

Multiple-Table Joins

A multiple-table join is a join of more than two tables. Its structure is similar to the structure of a two-table join, except that you have a join condition for more than one pair of tables in the WHERE clause. When columns from different tables have the same name, you must distinguish them by preceding the name with its associated table or table alias, as in *table.column*. See “Table Name” on page 7-434 for the full syntax of a table name.

The following multiple-table join yields the company name of the customer who ordered an item, as well as the stock number and manufacturer code of the item.

Figure 7-111

Example of a multiple-table join

```
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 two and one-half. The letters **x** and **y** are each aliases for the **stock** table.

Figure 7-112

Example of a self-join

```
SELECT x.stock_num, x.manu_code, y.stock_num, y.manu_code
FROM stock x, stock y
WHERE x.unit_price > 2.5 * y.unit_price
```

Outer Joins

The following outer join lists the company name of the customer and all associated order numbers, if the customer has placed an order. If not, the company name still is listed and a NULL value is returned for the order number.

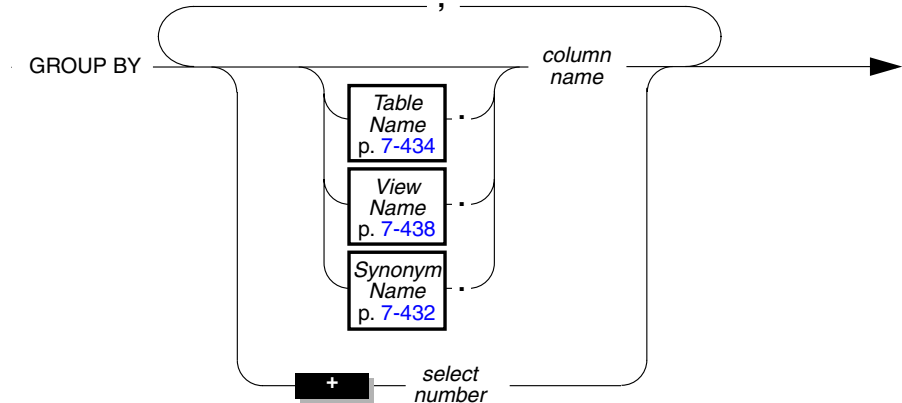
Figure 7-113
Example of an outer join

```
SELECT company, order_num
FROM customer c, OUTER orders o
WHERE c.customer_num = o.customer_num
```

See the *IBM Informix Guide to SQL: Tutorial* for more information about outer joins.

GROUP BY Clause

Use the GROUP BY clause to produce a single row of results for each group. A group is a set of rows that have the same values for each column listed.



column name is the name of a column or set of columns joined by a relational operator that is in the SELECT clause. Do not include columns of type BYTE or TEXT.

select number is an integer that represents the placement of a column or expression in the SELECT clause.

Using a GROUP BY clause restricts what you can enter in the SELECT clause. If you use a GROUP BY clause, each of the columns 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.

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.

Figure 7-114
Example of a GROUP BY clause

```
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 select 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 clause uses select numbers for **order_date** and **paid_date - order_date** in the GROUP BY clause. Note that you only can group by a combined expression by using the select-number notation. In the second SELECT clause, you cannot replace the 2 with the expression **paid_date - order_date**.

Figure 7-115
Examples of select numbers in a GROUP BY clause

```
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
```

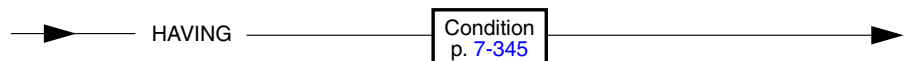
Select numbers (or column numbers) are required when SELECT statements are joined by UNION or UNION ALL keywords and different names for compatible columns appear in the same position.

Nulls in the GROUP BY Clause

Each row that contains a null value in a column specified by a GROUP BY clause is considered to belong to a single group. That is, all null values are grouped together.

HAVING Clause

Use the HAVING clause to apply one or more qualifying conditions to groups.



In the following examples, each condition compares one calculated property of the group with another calculated property of the group or with a constant. The first SELECT statement uses a HAVING clause that compares the calculated expression COUNT(*) with the constant 2. The query returns the average total price per item on all orders that have more than two items. The second SELECT statement lists customers and the call months if they have made two or more calls in the same month.

Figure 7-116

Using the HAVING clause with calculated values

```
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 query 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.

Figure 7-117

Using the HAVING clause with column values

```
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 query returns the average price of all the values in the table, as long as more than ten rows are in the table.

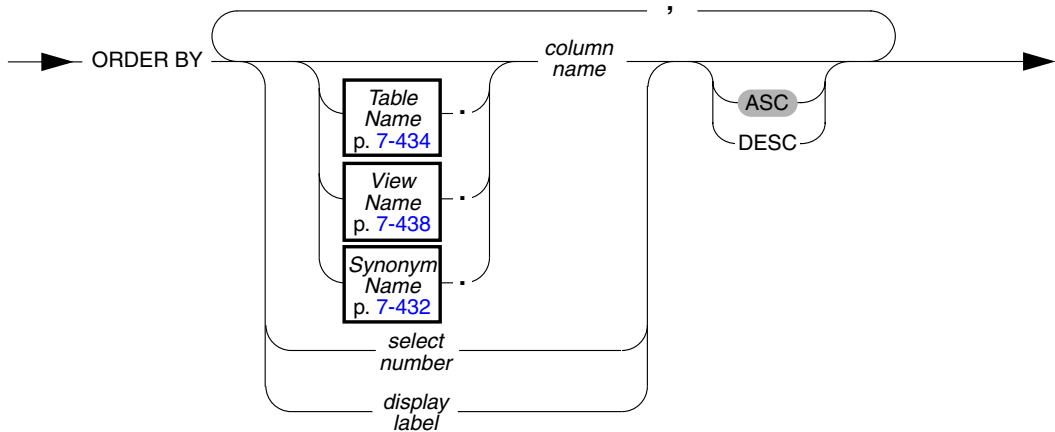
Figure 7-118

Using the HAVING clause without a GROUP BY clause

```
SELECT AVG(total_price) FROM items
HAVING COUNT(*) > 10
```


ORDER BY Clause

Use the ORDER BY clause to sort query results by the values contained in one or more columns.



column name is the name of a column from the SELECT clause by which you want to sort the query results.

display label is the display label used for a column or expression in the select list.

select number is an integer that represents the placement of a column or expression in the SELECT clause.

You can only order by columns, or expressions that contain column expressions, that are named in the SELECT clause. The sole exception is aggregate expressions; you cannot use aggregate expressions in the ORDER BY clause.

The following query explicitly selects the order date and shipping date from the **orders** table and then orders the query by the order date. By default, the query results are ordered in ascending order.

Figure 7-119

Using the ORDER BY clause when a column is selected explicitly

```
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 *`, so you can use `order_date` in the `ORDER BY` clause.

Figure 7-120

Using the ORDER BY clause when a column is selected implicitly

```
SELECT * FROM orders
ORDER BY order_date
```

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.

Figure 7-121

Ordering by a derived column

```
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 character data types, the ASCII collating sequence is used. See page [7-430](#) for a listing of the collating sequence.

Nulls in the ORDER BY Clause

Null values are ordered as less than non-null values. 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. For example, the following query selects all the rows in the `cust_calls` table, then orders them by `call_code` and by `call_dtime` within `call_code`.

Figure 7-122
A nested sort

```
SELECT * FROM cust_calls
ORDER BY call_code, call_dtime
```

Using Select Numbers

In the 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 example, the following query orders by the expression `paid_date - order_date` and `customer_num`.

Figure 7-123
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 UNION or UNION ALL keywords and compatible columns in the same position have different names.

ORDER BY Clause with DECLARE

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. ♦

I4GL

ESQL

INTO TEMP Clause

—▶ INTO TEMP — *temp table name* —————▶
 WITH NO LOG

temp table name is the simple name of a table. You cannot use any of the extended syntax described in the Table Name segment on page 7-434. You are limited to the conventions described in the Identifier segment on page 7-399.

The INTO TEMP clause creates a temporary table that contains the query results.

Temporary tables are always located in the root dbspace. The initial and next extents for the temp table are always eight pages.

The temporary table disappears when your program ends or when you issue a DROP TABLE statement on the temporary table. If your database does not have logging, or if it has logging and you created the temporary table without the WITH NO LOG keywords, the temporary table disappears when you close the current database.

If you use the same query results more than once, a temporary table saves you 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; the data is not updated as changes are made to the tables used to build the temporary table.

The column names of the temporary table are those named in the SELECT clause. You must supply a display label for all expressions other than simple column expressions. The display label for a column or expression becomes the column name in the temporary table. If you do not provide a display label for a column expression, the temporary table uses the column name from the select list. For example, the following query creates the **pushdate** table with two columns, **customer_num** and **slowdate**.

Figure 7-124
Creating a temporary table

```
SELECT customer_num, call_dtime + 5 UNITS DAY
FROM cust_calls INTO TEMP pushdate
```

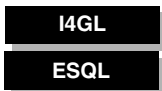
You can put indexes on the temporary table.

INTO TEMP Clause and INTO

Do not use the INTO option with the INTO TEMP clause. If you do, no results are returned to the program variables and the sqlcode variable is set to a negative value. The name of the sqlcode variable for each product is shown here.

4GL	ESQL/C	ESQL/COBOL
STATUS SQLCA.SQLCODE	sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA

◆



WITH NO LOG Option

If you use the WITH NO LOG keywords, operations on the temporary table are not included in the transaction log operations. You can use this option to reduce the overhead of transaction logging.

UNION Operator

Place the UNION operator between two SELECT statements to combine the queries into a single query. You can string several SELECT statements together using the UNION operator. Corresponding items do not need to have the same name.

Restrictions on a Combined SELECT

There are several restrictions on the queries that you can connect with a UNION operator:

- The number of items in the SELECT clause of each query must be the same, and the corresponding items in each SELECT clause must have compatible data types.
- If you use an ORDER BY clause, it must follow the last SELECT clause and you must refer to the item ordered by integer, not by identifier. Ordering takes place after the set operation is complete.
- You cannot use a UNION operator inside a subquery or in the definition of a view.
- You cannot use an INTO clause in a query unless you are sure that the compound query will return exactly one row and you are not using a cursor. In this case, the INTO clause must be in the first SELECT statement. ♦

You can put the results of a UNION operator into a temporary table by putting an INTO TEMP clause in the final SELECT statement.

The displayed column names are the column names or display labels from the first SELECT statement. ♦

I4GL**ESQL****ISQL**

Duplicate Rows in a Combined SELECT

If you use the UNION operator by itself, 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). For example, the following query 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 1990 and the first quarter of 1991.

Figure 7-125

Connecting two SELECT statements with a UNION ALL operator

```
SELECT customer_num, call_code FROM cust_calls
  WHERE call_dtime BETWEEN
    DATETIME (1990-1-1) YEAR TO DAY
    AND DATETIME (1990-3-31) YEAR TO DAY

UNION ALL

SELECT customer_num, call_code FROM cust_calls
  WHERE call_dtime BETWEEN
    DATETIME (1991-1-1) YEAR TO DAY
    AND DATETIME (1991-3-31) 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 just once. (If you want to remove duplicates within each SELECT statement, you would use the DISTINCT keyword in the SELECT clause, as described on page [7-260](#).)

Reference

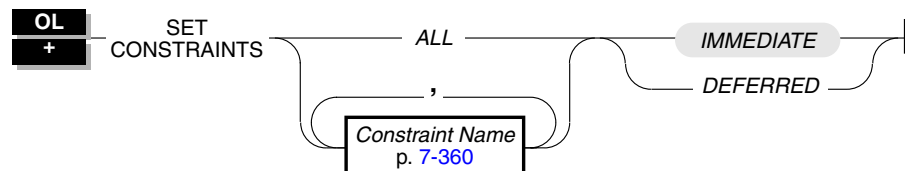
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of outer joins.

SET CONSTRAINTS

Purpose

Use the SET CONSTRAINTS statement to turn effective checking off and on in database with logging.

Syntax



Usage

The default constraint checking mode is IMMEDIATE. When the SET CONSTRAINTS statement is set to IMMEDIATE, effective checking is turned on and all specified constraints are checked at the end of each INSERT, UPDATE, or DELETE statement. If a constraint error occurs, the statement is not executed.

When you set the SET CONSTRAINTS statement to DEFERRED, effective checking is turned off and all specified constraints are not checked *until* the transaction is committed. If a constraint error occurs while the transaction is being committed, the transaction is rolled back. You only can set constraints to deferred in a database with logging.

The duration of the SET CONSTRAINTS statement is the transaction in which it is executed. You cannot execute the SET CONSTRAINTS statement outside of a transaction. Once a COMMIT or ROLLBACK WORK statement is successfully completed, the constraint mode of all constraints reverts to IMMEDIATE.

To revert from deferred to effective checking, you either can set the SET CONSTRAINTS to IMMEDIATE or use a COMMIT or ROLLBACK statement in your transaction.

You cannot explicitly defer the NOT NULL constraint for a column (NOT NULL constraints are not named) or set of columns. However, if you defer the checking of a primary key constraint, the checking of the NOT NULL constraint for that column or set of columns is also deferred. To defer the checking of all NOT NULL constraints, you must defer all constraints.

References

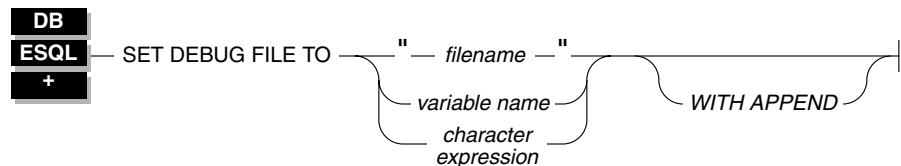
In this manual, see the CREATE TABLE statement.

SET DEBUG FILE TO

Purpose

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



character expression is any expression that evaluates to a usable filename.

filename is the full path and filename of the file that is to contain the trace statement output.

variable name is a character variable that contains the full path and filename of the file that is to contain the trace statement output.

Usage

This statement indicates that the output of the procedure TRACE statement goes to the file indicated by *filename*. The WITH APPEND option indicates to add to the file, if it exists. If you do not use the WITH APPEND keywords, the file is overwritten when you issue another SET DEBUG FILE TO statement with the same filename.

STAR

INET

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 another database server, the output file is located in your home directory on the machine of the database server. If you provide a full pathname for the debug file, the file is placed in the directory and file specified on the machine of the database server. If you do not have write permissions in the directory, an error is returned. ♦

To close the file opened by the SET DEBUG FILE TO statement, issue another SET DEBUG FILE TO statement with another filename. You can then edit the contents of the first file.

You can use the SET DEBUG FILE TO statement outside of 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.

The following example sends the output of the SET DEBUG FILE TO statement to a file called **debugging.out**:

```
SET DEBUG FILE TO "debugging" || ".out"
```

References

In this manual, see the section [“Debugging a Procedure” on page 8-14](#) and the TRACE statement on page [8-80](#).

In *IBM Informix Guide to SQL: Tutorial*, see the discussion of stored procedures.

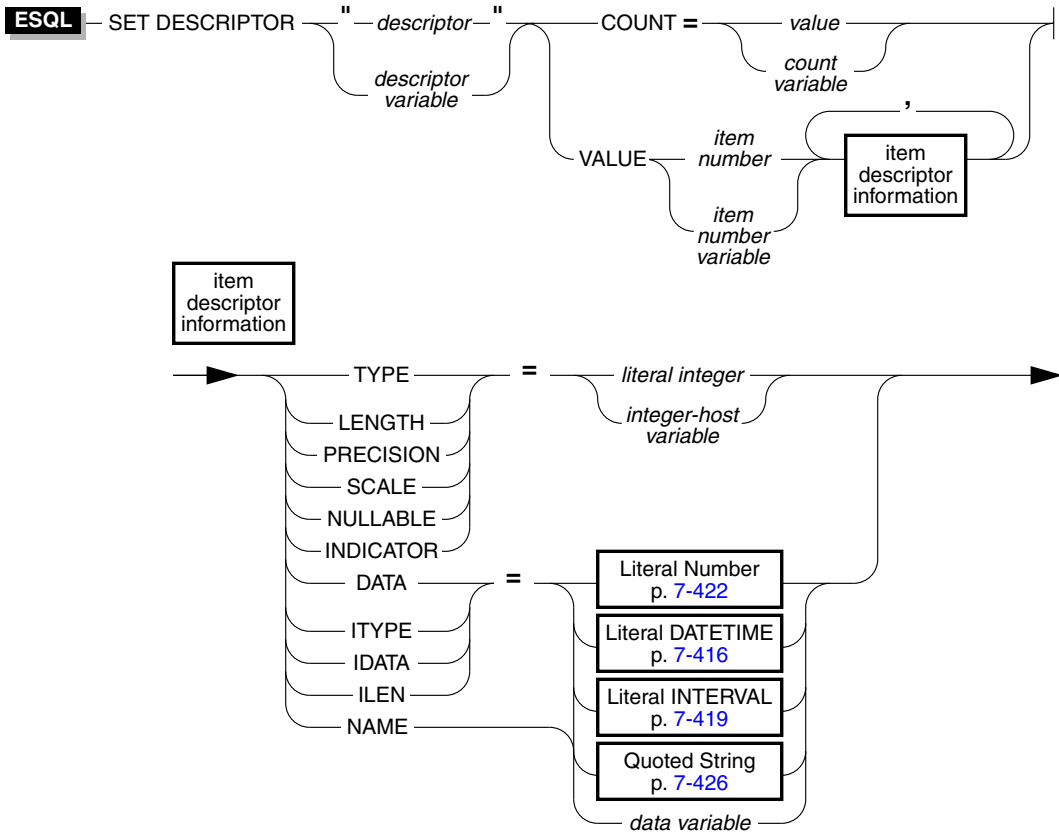
SET DESCRIPTOR

Purpose

Use the SET DESCRIPTOR statement to assign values to a system descriptor area in three different 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. (The items are typically those in a WHERE clause.)
- To set the item descriptor fields for each value for which you are providing descriptions in the system descriptor area. (The items are typically those 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.

Syntax



count variable is a variable that holds a literal integer that specifies how many items are being described in the system descriptor area. The value must be less than or equal to the number of occurrences in the system descriptor area, which is set when the area is allocated.

data variable is a host variable that contains the information appropriate for the field being set.

descriptor is a string that identifies a currently allocated system descriptor area.

<i>descriptor variable</i>	is an embedded variable name that contains a string that identifies a currently allocated system descriptor area.
<i>integer host variable</i>	is the name of a variable that contains an integer value that is appropriate for the indicated field. For the TYPE field, the correspondence between the integer codes and data types is provided in Figure 7-127 on page 7-296 .
<i>item number</i>	is an unsigned integer that represents one of the items in the descriptor area.
<i>item number variable</i>	is the name of an integer host variable that contains an unsigned integer that represents one of the items in the descriptor area.
<i>literal integer</i>	is a positive, nonzero integer that represents the data type of the item. The correspondence between the integer codes and data types is provided in Figure 7-127 on page 7-296 .
<i>value</i>	is a literal integer that specifies how many items are being described in the system descriptor area. The value must be less than or equal to the number of occurrences in the system descriptor area, which is set when the area is allocated.

Usage

If an error occurs during the assignment to any of the identified system descriptor fields, the contents of all identified fields are set to zero or null, depending on the type of that variable.

COUNT Option

Use the COUNT option to set the number of items that are to be used in the system descriptor area.

If you allocated 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 actually are using. For example, the sequence of statements in [Figure 7-126](#) (shown using IBM Informix ESQL/C) can be used in a program.

Figure 7-126

Example of setting the COUNT field of a system descriptor using ESQL/C

```
EXEC SQL BEGIN DECLARE SECTION;
INT count, itemno, type, length;
CHAR chval[21];
EXEC SQL END DECLARE SECTION;

ALLOCATE DESCRIPTOR 'desc_100'; /*allocates for 100 items*/

count = 2;
EXEC SQL SET DESCRIPTOR 'desc_100' COUNT = :count;
```

VALUE Option

Use the VALUE option 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 have been described by the database server during a DESCRIBE statement.

Setting the TYPE Field

Use the set of codes shown in [Figure 7-127](#) to set the value of TYPE for each item.

Figure 7-127

List of data type integer constants

SQL Data Type	Integer Value
CHAR	0
SMALLINT	1
INTEGER	2
FLOAT	3
SMALLFLOAT	4
DECIMAL	5
SERIAL	6
DATE	7

(1 of 2)

SQL Data Type	Integer Value
MONEY	8
DATETIME	10
BYTE	11
TEXT	12
VARCHAR	13
INTERVAL	14

(2 of 2)

Figure 7-128 shows how you can set the TYPE field in ESQL/C.

Figure 7-128
Examples of setting TYPE using ESQL/C

```
main()
{
  $int count, itemno, type, length;
  ...
  $ALLOCATE DESCRIPTOR 'desc1' WITH MAX 5;
  ...
  $SET DESCRIPTOR "desc1" VALUE 2 TYPE = 5;

  type = 2; itemno = 3;
  $SET DESCRIPTOR "desc1" VALUE $itemno TYPE = $type;
}
```

If you do not compile using the **-xopen** option, the regular Informix SQL code is assigned for TYPE. You must be careful not to mix normal and X/Open modes because errors can result. For example, if a particular type is not defined under X/Open mode but is defined under normal mode, execution of a SET DESCRIPTOR statement can result in an error.

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 X/Open codes for data types are shown in [Figure 7-129](#).

Figure 7-129
X/Open data type codes

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. ♦

Setting the DATA Field

When you set the DATA field, you must provide the appropriate type of data (character string for CHAR or VARCHAR, integer for INTEGER, and so on).

When any value other than DATA is set, the value of DATA is undefined. You cannot set the DATA field for an item without setting TYPE for that item. If you set the TYPE field for an item to a character type, you must set the LENGTH field as well. If you do not set the LENGTH field for a character item, you receive an error.

Using DECIMAL or MONEY Types

If you set the TYPE field for a DECIMAL or MONEY 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 for the SCALE and PRECISION supplied.

Using DATETIME or INTERVAL 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.

E/C

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 type DATETIME or INTERVAL, you do not need to set LENGTH explicitly to the encoded qualifier integer. ♦

E/CO

To determine the encoded qualifiers for a DATETIME or INTERVAL character string, use the ECO-IQL routine. ♦

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, indicating 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 in [Figure 7-127](#).

Modifying Values Set by the DESCRIBE Statement

You can modify the contents of a system descriptor area after it is set by a DESCRIBE statement.

E/CO

After you use a DESCRIBE statement on SELECT or an INSERT statement, you must check to determine whether the TYPE field is set to either 11 or 12, indicating a TEXT or BYTE data type. If TYPE contains an 11 or a 12, you must use the SET DESCRIPTOR statement to reset the TYPE to 116 to indicate FILE type. ♦

References

In this manual, for further information about using dynamic SQL statements, see the following statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, and PUT.

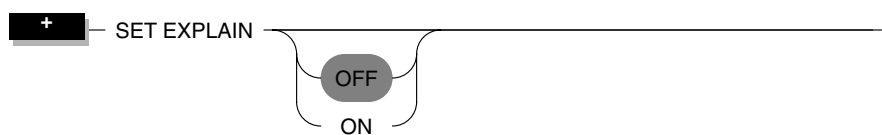
For further information about the system descriptor area, see [Chapter 6, "Using Descriptors."](#)

SET EXPLAIN

Purpose

Use the SET EXPLAIN statement to obtain a measure of the work involved in performing a query.

Syntax



Usage

The SET EXPLAIN statement executes during the database server optimization phase, which occurs when a query is initiated. For queries associated with a cursor, the query is initiated when you open the cursor.

When you issue a SET EXPLAIN ON statement, the access procedures of all subsequent queries and the path chosen by the optimizer are passed forward and stored in your current directory in a file with the name **sqexplain.out**. If the file already exists, subsequent output is appended to it.

The SET EXPLAIN ON statement remains in effect until you issue another SET EXPLAIN statement or until the program ends.

SET EXPLAIN Output

The SET EXPLAIN output file contains a copy of the query, a plan of execution as selected by the database server optimizer, and an estimate of the amount of work to be done. The optimizer selected this plan as the most efficient way of performing the query, based on such things as the presence and type of indexes and the number of rows in each table.

The estimate of work is a weighted sum that you can use to compare the cost of one path to another. The work estimate is in arbitrary units. A single disk access is one unit and other actions are scaled to that. However, the sum does not translate directly into time and there is no direct correlation between the work involved in a query and the resources used. It is generally true that a query with a higher estimate is likely to take longer to run than one with a smaller estimate.

In addition to the number estimate, the output file also contains the following information:

- A guess at the number of rows to be returned
- The order in which the optimizer accesses tables
- One of the following methods (access paths) by which the optimizer reads each table:

SEQUENTIAL SCAN	Reads rows in sequence
INDEX PATH	Scans one or more indexes
AUTOINDEX PATH	Creates a temporary index
SORT SCAN	Sorts the result of the preceding join or table scan
MERGE JOIN	Uses a sort-merge join instead of nested-loop join
- The table column or columns that serve as a filter, if any, and whether the filtering is through an index.

The name of the table owner precedes table names in the output file.

Information returned regarding queries that include joins can be inaccurate. If the columns in the joins are indexed, the information is more reliable but still subject to inaccuracy.

The optimizer chooses the best path of execution to produce the fastest possible table join: either nested-loop join, sort-merge join, or a combination of the two. When a sort-merge join is used, two lines appear in the output file beginning with the words SORT SCAN and MERGE JOIN.

The SORT SCAN section indicates that a sort of the result of the preceding join or table scan is to be done in preparation for a sort-merge join. It includes a list of the columns that form the sort key. The order of the columns is the order of the sort. As with indexes, the default order is *ascending*. Where possible, this ordering is arranged to support any requested ORDER BY or GROUP BY clause.

The MERGE JOIN section indicates that a sort-merge join, instead of the (implied) nested-loop join, is to be used on the preceding join/table pair. It includes a list of the filters that control the sort-merge join, and, where applicable, a list of any other join filters. For example, a join of tables A and B with the filters $A.c1 = B.c1$ and $A.c2 < B.c2$ lists the first under “Merge Filters” and the second under “Other Join Filters.”

STAR

INET

If you invoke a SET EXPLAIN statement on your home machine, the output file is located in your current directory. If your current database is on another machine, the output file still goes to your local machine.

Also, the output file can contain a sixth type of table access, as follows:

REMOTE PATH Access another database (distributed databases only).

◆

SE

The IBM Informix SE database server generates fewer query-processing statistics than are available from the IBM Informix OnLine database server. As a result, estimates for the cost and the number of rows returned might be more precise if you are using IBM Informix OnLine than if you are using IBM Informix SE. Estimates returned for queries that include joins tend to be highly inaccurate. ◆

The following output examples represent what you might see when a SET EXPLAIN ON statement is issued using IBM Informix OnLine.

The first two examples contain two entries for a multiple-table query and show the SORT SCAN and MERGE JOIN lines. Note that in both cases, if SORT MERGE had not been chosen, the second table would have been scanned using an *autoindex path*.

Figure 7-130

Sample output for multiple-table query and MERGE JOIN

```

QUERY:
-----
select i.stock_num from items i, stock s, manufact m
  where i.stock_num = s.stock_num
     and i.manu_code = s.manu_code
     and s.manu_code = m.manu_code

Estimated Cost: 52
Estimated # of Rows Returned: 130

1) rdtest.m: SEQUENTIAL SCAN

SORT SCAN: rdtest.m.manu_code

2) rdtest.s: SEQUENTIAL SCAN

SORT SCAN: rdtest.s.manu_code

MERGE JOIN:
  Merge Filters: rdtest.m.manu_code = rdtest.s.manu_code

3) rdtest.i: INDEX PATH

(1) Index Keys: stock_num manu_code
    Lower Index Filter: (rdtest.i.stock_num = rdtest.s.stock_num AND
rdtest.i.manu_code = rdtest.s.manu_code)

QUERY:
-----
select stock.description from stock, stock2
  where stock.description = stock2.description
     and stock.unit_price < stock2.unit_price

Estimated Cost: 15
Estimated # of Rows Returned: 370

1) rdtest.stock: SEQUENTIAL SCAN

SORT SCAN: rdtest.stock.description

2) rdtest.stock2: SEQUENTIAL SCAN

SORT SCAN: rdtest.stock2.description

MERGE JOIN
  Merge Filters: rdtest.stock2.description = rdtest.stock.description
  Other Join Filters: rdtest.stock.unit_price < rdtest.stock2.unit_price

```

The following sample output contains entries for a simple query and a complex query from the **customer** table.

Figure 7-131*Results of SET EXPLAIN for a simple and a complex query*

```

QUERY:
-----
SELECT fname, lname, company FROM customer

Estimated Cost: 3
Estimated # of Rows Returned: 28

1) joe.customer: SEQUENTIAL SCAN

QUERY:
-----
SELECT fname, lname, company FROM customer
   WHERE company MATCHES "Sport*" AND customer_num BETWEEN 110 AND 115
   ORDER BY lname;

Estimated Cost: 4
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

1) joe.customer: INDEX PATH

Filters: joe.customer.company MATCHES 'Sport*'

(1) Index Keys: customer_num
Lower Index Filter: joe.customer.customer_num >= 110
Upper Index Filter: joe.customer.customer_num <= 115

```

The following sample output is from an output file with an entry for a multiple-table query.

Figure 7-132*Results of SET EXPLAIN for a multiple-table query*

```

QUERY:
-----
SELECT * FROM customer, orders, items
   WHERE customer.customer_num = orders.customer_num
   AND orders.order_num = items.order_num

Estimated Cost: 20
Estimated # of Rows Returned: 69

1) joe.orders: SEQUENTIAL SCAN

2) joe.customer: INDEX PATH

(1) Index Keys: customer_num
Lower Index Filter: joe.customer.customer_num = joe.orders.customer_num

3) joe.items: INDEX PATH

(1) Index Keys: order_num
Lower Index Filter: joe.items.order_num = joe.orders.order_num

```

Reference

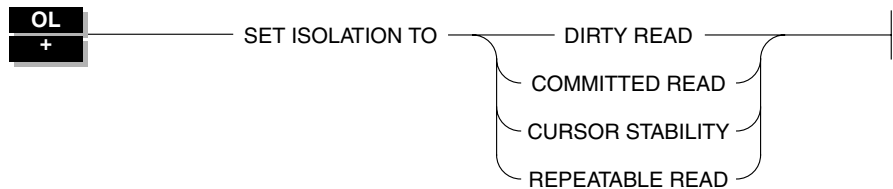
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of SET EXPLAIN.

SET ISOLATION

Purpose

Use the SET ISOLATION statement with the IBM Informix OnLine database server to define the degree of concurrency among processes that attempt to access the same rows simultaneously.

Syntax



Usage

The database isolation level affects concurrency when rows are retrieved from the database. IBM Informix OnLine uses shared locks to support four levels of isolation among processes attempting to access data.

The default isolation level for a particular database is established when you create the database, according to database type. The default isolation level for each database type follows:

- 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 you enter another SET ISOLATION statement or until the end of the program.

The level of isolation does not interfere with processes that are updating or deleting data. The update or delete process always acquires an exclusive lock on a row that is being modified.

Isolation Levels

The following definitions explain the critical characteristics of each isolation level. The isolation levels are listed in order, from the lowest level of isolation to the highest.

DIRTY READ Provides zero isolation. Dirty Read is appropriate for static tables that are used for queries. With a Dirty Read isolation level, it is possible for a query to return a *phantom row*; that is, an uncommitted row that was inserted or modified within a transaction which 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 row retrieved is committed in the table at the time that the row is retrieved. Even so, no locks are acquired. While one process uses a 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.

CURSOR STABILITY	Acquires a shared lock on the selected row. Another process also can 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, IBM Informix OnLine releases the shared lock.
REPEATABLE READ	Acquires a shared lock on every row selected during the transaction. Another process also can acquire a shared lock on a selected row, but no other process can modify any selected row during your transaction. If you repeat the query during the transaction, you reread the same information. The shared locks are released only when the transaction is committed or rolled back. Repeatable Read is the default isolation level in an ANSI-compliant database.

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.

The data obtained during Binary Large Object (BLOB) retrieval can vary, depending on the database isolation level. Under Dirty Read or Committed Read levels of isolation, a process is permitted to read a BLOB that is either deleted (if the delete is not yet committed), or in the process of being deleted. That is, under these isolation levels, certain conditions exist under which an application can read a deleted BLOB.

IBM Informix SQL and DB-Access do not use explicit cursors. The Committed Read and Cursor Stability isolation levels appear to have the same effect, although retrieval might be slower if you invoke Cursor Stability. ♦

You can issue a SET ISOLATION statement from a client machine only after a database has been opened. ♦

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.

ISQL

STAR

INET

I4GL

ESQL

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 Repeatable Read are released when the transaction is completed, but the scroll cursor with hold remains open beyond the end of the transaction. Thus, you can modify released rows as soon as the transaction ends, creating the possibility that the retrieved data in the temporary table can become inconsistent with the actual data. ♦

References

In this manual, see the following statements: CREATE DATABASE and SET LOCK MODE.

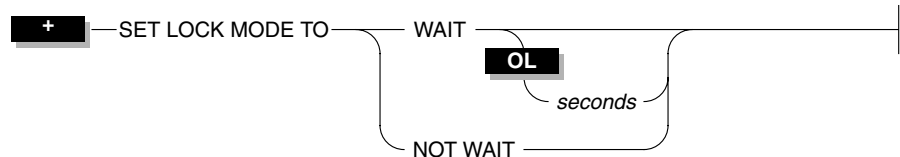
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of isolation levels.

SET LOCK MODE

Purpose

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



seconds is the maximum number of seconds that a process waits for a lock to be released.

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:

- NOT WAIT End the operation immediately and return an error code. This is the default condition.
- WAIT Suspend the process until the lock is released.
- WAIT *seconds* Suspend the process until the lock is released or until the end of a waiting period, specified in seconds. If the lock remains after waiting, end the operation and return an error code.

SE

IBM Informix SE does not support the *seconds* option. If you decide that a process should wait for a lock to be released, you cannot limit the waiting period. ♦

SE

The lock mode has no effect with respect to exclusive locks. Whenever a process attempts to access a row, table, or database that is locked in exclusive mode, the operation ends and an error code is returned.

The SET LOCK MODE option is available on machines that use kernel locking. To determine whether your machine uses kernel locking, inspect the directory that holds the database files. If the directory contains files with the extension **.lok**, your system does not use kernel locking and the SET LOCK MODE option is unavailable. ♦

WAIT Keyword

The database server protects you against the possibility of a deadlock when you request the WAIT option. Before suspending a process, the database server checks whether suspending the process could create a deadlock. If the database server discovers 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 deadlock or an actual deadlock, the database server returns an error.

Use with caution the unlimited waiting period created when the WAIT option is specified without *seconds*. If no upper limit is specified and the process that placed the lock somehow fails to release it, suspended processes could wait indefinitely. Since this is not a true deadlock situation, the database server does not take corrective action.

You can issue a SET LOCK MODE statement from a client machine only after a database has been opened. ♦

STAR

INET

STAR

The DBA establishes a default value for *seconds* that applies to your entire system. If you use a SET LOCK MODE statement to set your own upper limit, your value applies only when your waiting period is shorter than the system default. ♦

References

In this manual, see the following statements: LOCK TABLE, UNLOCK TABLE, and SET ISOLATION MODE.

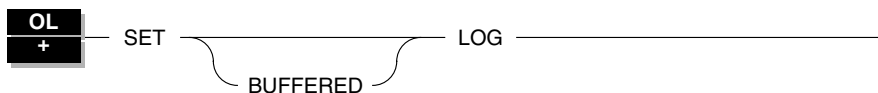
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of SET LOCK MODE.

SET LOG

Purpose

Use the SET LOG statement to change your IBM Informix OnLine database logging mode from buffered transaction logging to unbuffered transaction logging, or vice versa.

Syntax



Usage

You activate transaction logging when you create a database or add logging to an existing database. These transaction logs either can be buffered or unbuffered.

The default condition for transaction logs is unbuffered logging. As soon as a transaction ends, the IBM Informix OnLine database server writes the transaction to the disk. If a system failure occurs when you are using unbuffered logging, you recover all completed transactions.

You gain a marginal increase in efficiency with buffered logging, but you incur some risk. In the event of a system failure, the IBM Informix OnLine database server cannot recover the completed transactions that were buffered in memory.

The SET LOG statement changes the transaction logging mode to unbuffered logging; the SET BUFFERED LOG statement changes the mode to buffered logging.

The SET LOG statement redefines the mode for the current session only. The default mode, which the DBA sets using the DB-Monitor, remains unchanged.

STAR

The buffering option does not affect retrievals from external tables. A database with logging only can access other databases with logging, but it makes no difference whether the databases use buffered or unbuffered logging. ♦

ANSI

An ANSI-compliant database cannot use buffered logs. ♦

References

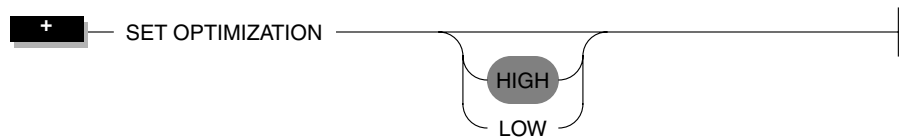
In this manual, see the following statements: CREATE DATABASE and START DATABASE.

SET OPTIMIZATION

Purpose

Use the SET OPTIMIZATION statement to specify a high or low level of database server optimization.

Syntax



Usage

You can execute a SET OPTIMIZATION statement at any time. The optimization level carries across databases but applies only within the current database server.

After a SET OPTIMIZATION statement executes, the new optimization level remains in effect until you enter another SET OPTIMIZATION statement or until the program ends.

The default database server optimization level, HIGH, remains in effect until you issue another SET OPTIMIZATION statement. The LOW option invokes a less sophisticated but faster optimization algorithm.

The algorithm invoked by a SET OPTIMIZATION HIGH statement is a sophisticated, cost-based strategy that examines all reasonable choices and selects the best overall alternative. For large joins, this algorithm can incur more overhead than desired. In extreme cases, you can run out of memory.

STAR

INET

The alternative algorithm invoked by a SET OPTIMIZATION LOW statement eliminates unlikely join strategies during the early stages and thus reduces the amount of time and resources spent during optimization. However, by specifying a low level of optimization, you take the risk that the optimal strategy is not selected because it was eliminated from consideration during early stages of the algorithm.

The following examples show optimization across a network. The **central** database (on machine 1) is to have LOW optimization; the **western** database (on machine 2) is to have HIGH optimization. If the **western** database were on the same machine as **central**, it would have LOW optimization.

Figure 7-133

Example of a SET OPTIMIZATION statement across an IBM Informix STAR network

```
set optimization low;
database central;
select * from stock;
close database;
database western@rockies;
select * from stock;
```

Figure 7-134

Example of a SET OPTIMIZATION statement across an IBM Informix NET network

```
set optimization low;
database central;
select * from stock;
close database;
database //rockies/western;
select * from stock;
♦
```

References

In *IBM Informix Guide to SQL: Tutorial*, see the discussion of optimizing queries.

START DATABASE

Purpose

Use the START DATABASE statement with an IBM Informix SE database server to start recording transactions, to make a database ANSI-compliant or to change the name of an existing transaction log file.

Syntax

```

SE
+
START DATABASE Database Name WITH LOG IN "pathname"


MODE ANSI


```

Database Name p. 7-362

pathname is the pathname of the transaction log file, enclosed in quotation marks.

Usage

To use the START DATABASE statement, all of the following conditions must be true:

- You have DBA privilege.
- There is no current database.
- The directories specified in *pathname* exist.

For maximum protection, specify a location for the transaction log that is not on the same storage device as the database.

Issue a CLOSE DATABASE statement before you create and start a transaction log. The START DATABASE statement locks the database exclusively to prevent access by other processes. If another process is using the database (even if the database is only being read), the START DATABASE statement fails.

The database remains locked after the START DATABASE statement executes. When you are satisfied that the database is ready for use, remove the exclusive lock by executing the CLOSE DATABASE statement. Reopen the database with the DATABASE statement.

MODE ANSI Keywords

Use the MODE ANSI keywords to make a database ANSI-compliant. An ANSI-compliant database conforms to different transaction-processing and object-naming conventions than does a database that does not comply with ANSI standards.

The following example starts an ANSI-compliant database named **stores5**:

```
START DATABASE stores5
WITH LOG IN "/u/myname/stores5.log" MODE ANSI
♦
```

Transaction Log Name Change

You must issue a START DATABASE statement immediately before you archive the database if you plan to change the name or the location of the transaction log. Specify the new path to the transaction log in the START DATABASE statement.

References

In this manual, see the following statements: CREATE DATABASE and ROLLFORWARD DATABASE.

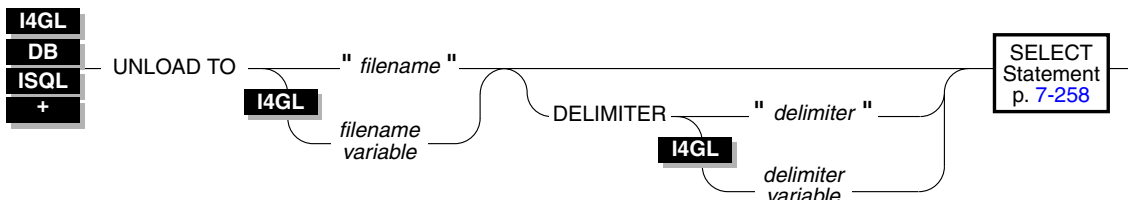
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of transaction processing.

UNLOAD

Purpose

Use the UNLOAD statement to write the rows retrieved in a SELECT statement to an ASCII operating system file.

Syntax



delimiter is a quoted character that serves as the delimiter between fields. The default delimiter is the vertical bar (| = ASCII 124) or the value of the **DBDELIMITER** environment variable, if set.

delimiter variable is a character variable that contains the character to use as the delimiter between fields. The default delimiter is the vertical bar (| = ASCII 124) or the value of the **DBDELIMITER** environment variable, if set.

filename is a quoted string constant that specifies the name of a file. Rows retrieved in the SELECT statement are written to this file.

filename variable is a CHARACTER or VARCHAR variable that contains a file name. Rows retrieved in the SELECT statement are written to this file.

Usage

To use the UNLOAD statement, you must have Select privileges on all columns selected in the SELECT statement. For information on database-level and table-level privileges, see the GRANT statement on page 7-175.

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 7-258.)

You cannot use the PREPARE statement to preprocess an UNLOAD statement. ♦

UNLOAD TO File

The UNLOAD TO file contains the selected rows retrieved from the table.

The following table describes how IBM Informix 4GL, IBM Informix SQL, and DB-Access format the output.

Figure 7-135
Types of data and their output format for an UNLOAD statement

Type of Data	Output Format
character	If a character field contains the delimiter character, IBM Informix products automatically escape it with a backslash to prevent interpretation as a special character. (If you use a LOAD statement to insert the rows into a table, backslashes are automatically stripped.) 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, unless the DBDATE environment variable has been set and another format specified.
MONEY	MONEY values are unloaded with no leading currency symbol.
NULL	NULL columns are unloaded by placing no characters between the delimiters.

(1 of 2)

Type of Data	Output Format
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.

(2 of 2)

Do not use the backslash character 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 containing VARCHAR or BLOB 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 thus very difficult to print or edit.
- Trailing blanks are retained in VARCHAR fields.
- Do not use the following characters as delimiting characters in the UNLOAD TO file: 0-9, a-f, A-F, space, tab, or backslash.

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**, looks as follows:

```
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, IBM Informix 4GL, IBM Informix SQL, and DB-Access check the **DBDELIMITER** environment variable.

If the **DBDELIMITER** variable has not been set, the default delimiter is the vertical bar (| = ASCII 124). See [Chapter 4, “Environment Variables,”](#) for information about how to set the **DBDELIMITER** environment variable.

The following statement specifies the semicolon (;) as the delimiter character:

```
UNLOAD TO "cust.out" DELIMITER ";"
SELECT fname, lname, company, city
FROM customer
```

References

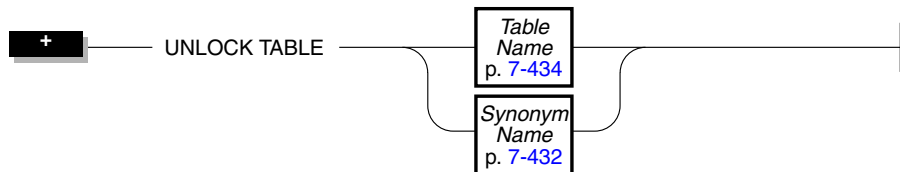
In this manual, see the following statements: LOAD and SELECT.

UNLOCK TABLE

Purpose

Use the UNLOCK TABLE statement in a database without transactions to unlock a table that you previously locked with the LOCK TABLE statement.

Syntax



Usage

You can lock a table if you own the table or if you have Select privileges on the table, either from a direct grant or from a grant to PUBLIC. You only can unlock a table that you locked. You cannot unlock a table that was locked by another process.

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.

Only one lock can apply to a table at a time.

To change the lock mode of a table in a database without transactions, you first unlock the table using the UNLOCK TABLE statement, then issue a new LOCK TABLE statement.

The UNLOCK TABLE statement fails if it is issued within a transaction. Table locks set within a transaction are released automatically when the transaction is completed.

You should not issue an UNLOCK TABLE statement within an ANSI-compliant database. The UNLOCK TABLE statement fails if it is issued within a transaction, and a transaction is always in effect in an ANSI-compliant database. ♦

ANSI

References

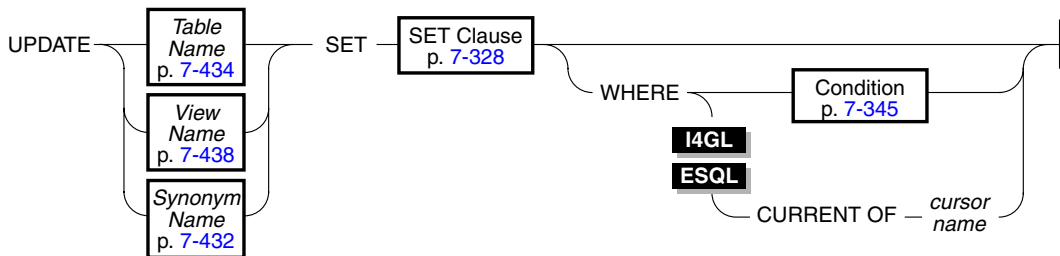
In this manual, see the following statements: COMMIT WORK, ROLLBACK WORK, and LOCK TABLE.

UPDATE

Purpose

Use the UPDATE statement to change the values in one or more columns of one or more rows in a table or view.

Syntax



cursor name is the cursor identifier.

Usage

To update data in a table, you must either own the table or have Update privileges for the table (see the GRANT statement on page 7-175). To update data in a view, you must have the required Update privileges, and the view must meet the requirements explained in the section “Updating Rows Through a View” on page 7-326.

If you omit the WHERE clause, all rows of the target table are updated.

If you omit the WHERE clause and are in interactive mode, IBM Informix SQL and DB-Access do 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 IBM Informix SQL or DB-Access from the command line, the statement executes immediately. ♦

DB

ISQL

Updating Rows Through a View

You can update data through a *single-table* view if you have Update privileges on the view (see the GRANT statement on page 7-175). 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 referred to as a virtual column)
- Aggregate value

You can use data integrity constraints to prevent users from updating values into the underlying table that do not fit the view-defining SELECT statement. For further information, refer to the WITH CHECK OPTION discussion in the CREATE VIEW statement on page 7-97.

Since duplicate rows can occur in a view even though the underlying table has unique rows, be very 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 were to update one of the two duplicate total price values, you would have no way of knowing which item price was updated.

Updating Rows in a Database Without Transactions

If you are updating rows in a database without transactions, you must take explicit action to restore updated rows. For example, if the UPDATE statement fails after updating some rows, the successfully updated rows remain in the table. You cannot automatically recover from a failed update.

Updating Rows in a Database with Transactions

If you are updating rows in a database with transactions and you are using transactions, you can undo the update using the ROLLBACK WORK statement. If you do not execute a BEGIN WORK statement before the update and the update fails, the database server automatically rolls back any database modifications made since the beginning of the update.

ANSI

If you are updating rows in an ANSI-compliant database, transactions are implicit and all database modifications take place within a transaction. In this case, if an UPDATE statement fails, you can use the ROLLBACK WORK statement to undo the update. ♦

Locking Considerations

If you are using an IBM Informix OnLine database server, 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 not to update or delete it. Just before the update occurs, the update process *promotes* the shared lock to an exclusive lock. An exclusive lock prevents other processes from reading or modifying the contents of the row until the lock is released.

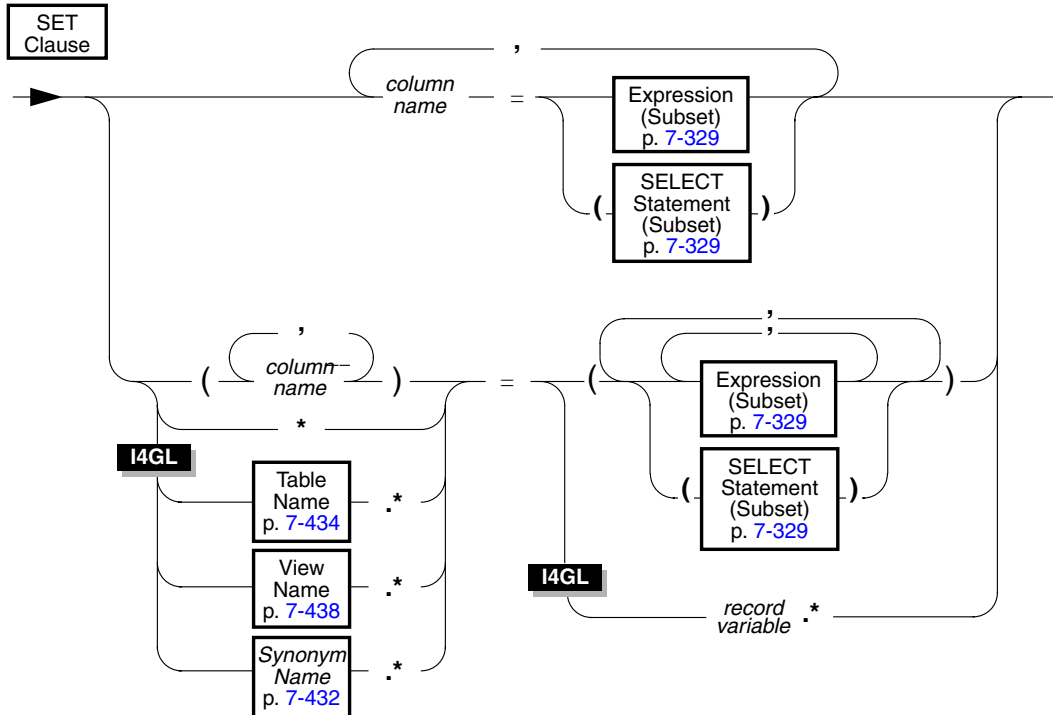
IBM Informix OnLine allows only one update lock at a time on a row or a page (the type of lock depends on the lock mode selected in the CREATE TABLE or ALTER TABLE statements). An update process can acquire an update lock on a row or a page that has a shared lock from another process, but you cannot promote the update lock from shared to exclusive (and the update cannot occur) until the other process releases its lock.

If the number of rows affected by a single update is very large, you can exceed the limits placed on the maximum number of simultaneous locks. If this occurs, you can reduce the number of transactions per UPDATE statement or lock the page (IBM Informix OnLine database servers only) or the entire table before you execute the statement.

SE

Individual rows of a table are locked automatically when you execute an UPDATE statement. This feature increases the likelihood that selecting processes running on IBM Informix SE will encounter exclusive locks on rows. As a result, you might experience reduced concurrency during processing. ♦

SET Clause



* indicates that you want to update all columns in *table name*.

column name names the columns that you want to update. You cannot update SERIAL data type columns.

record variable indicates that the columns are to be updated with values contained in an 4GL program record.

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.

Selecting All Columns with the Set Clause

You can use the `.*` extension with *synonym name*, *table name*, *view name*, or *record variable* to indicate that you want to select all columns for update. ♦

Subset of Expressions Allowed in the SET Clause

You cannot use an expression made up of aggregate functions in the SET clause. For a complete description of syntax and usage, see the Expression segment on page 7-370.

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 7-258.

Single Columns Paired to Single Expressions

You can include any number of single-column, single-expression pairs in an UPDATE statement.

The following examples illustrate the single-column to single-expression form of the SET clause.

Figure 7-136

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
```

Multiple Columns Equal to Multiple Expressions

The SET clause offers you two options for listing a series of columns you intend to update:

- Explicitly list each column, separating by commas and enclosing all in parentheses.
- Implicitly list all columns in *table name* using the asterisk notation (*).

To complete the SET clause, you must list each expression explicitly, separated by commas and all enclosed 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.

Figure 7-137

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),
     "07/01/1990")
  WHERE col4 = 1001
```


IBM Informix 4GL provides an alternative to the implicit asterisk notation (*), which is *table name.**. IBM Informix 4GL also provides a second option for listing the expressions that complete the SET clause. You can set the listed columns equal to the full set of values contained in a 4GL record variable with the *record variable.** notation.

You can combine these additional elements with other SET clause options in an 4GL UPDATE statement, as long as the statement adheres to syntax rules. If you want to update all columns in a table that contain a SERIAL column, you can use the record variable form of the SET clause even though you cannot update SERIAL columns. When 4GL executes an UPDATE statement that contains a record variable in the SET clause, it automatically skips any SERIAL column and its corresponding value in the expression list produced by *record variable.**. ♦

The following 4GL example illustrates this form of the SET clause.

Figure 7-138

Example of a record variable in the SET clause in IBM Informix 4GL

```

DEFINE
    p_cust RECORD LIKE customer.*
    tmp_cust RECORD LIKE customer.*

LET upd_stmt = "SELECT * FROM customer ",
               "WHERE customer_num = ? FOR UPDATE"
PREPARE upd_prep FROM upd_stmt
DECLARE upd_cur CURSOR FOR upd_prep

{ . . . Prompt here for p_cust search criteria. }

OPEN upd_cur USING p_cust.customer_num
FETCH upd_cur INTO p_cust.*

IF STATUS = NOTFOUND THEN
    ERROR "No entry for customer number ", p_cust.customer_num, "."
ELSE
    LET tmp_cust.* = p_cust.* { saves current values in case }
                             { user aborts the update           }

    { . . . Prompt here for new customer table values. }

    LET int_flag = false
    INPUT BY NAME p_cust.fname thru p_cust.phone
    WITHOUT DEFAULTS

    IF NOT int_flag THEN { User did not press Interrupt key. }

        UPDATE customer SET customer.* = p_cust.*

```

```
        ELSE { int_flag is true: user pressed INTERRUPT key }
            LET int_flag = false
            LET p_cust.* = tmp_cust.*
            ERROR "Update cancelled."
        END IF { int_flag test}
    END IF { status = notfound on upd_cur }
```

WHERE Clause

The WHERE clause allows you to 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 7-258). 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**.

Figure 7-139

Using a WHERE condition within an UPDATE statement

```
UPDATE customer
  SET (state, zipcode, phone) =
    (SELECT state, zipcode, phone FROM new_address
     WHERE new_address.cust_num =
       customer.cust_num)
  WHERE customer.cust_num IN
    (SELECT cust_num FROM new_address)
```

4GL

ESQL

WHERE CURRENT OF Clause

You can use the CURRENT OF keywords 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 only can 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 keywords if you associate the UPDATE statement with a cursor that was created with the FOR UPDATE keywords. (See the DECLARE statement on page 7-107.) If the cursor was created without specifying any columns for update, 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. IBM Informix SE and IBM Informix OnLine usually can perform updates more quickly if columns are specified in the DECLARE statement. ♦

The following IBM Informix 4GL 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. The UPDATE statement is prepared outside the WHILE loop to ensure that parsing is done only once. (For more information, refer to the PREPARE statement on page 7-218.)

Figure 7-140
Example of the WHERE CURRENT OF form of the WHERE clause in an IBM Informix 4GL program

```

PREPARE sel_stmt FROM
  "SELECT * FROM customer ",
  "WHERE cust_num between ? and ? FOR UPDATE"
DECLARE x CURSOR FOR sel_stmt
OPEN x USING low, high

PREPARE u FROM
  "UPDATE customer SET discount = 0.1",
  "WHERE CURRENT OF x"

WHILE TRUE
  FETCH x INTO r_cust.*
  IF STATUS = NOTFOUND THEN
    EXIT WHILE
  END IF

```



```
LET ptext = "Update ", r_cust.fname CLIPPED, " ",
           r_cust.lname CLIPPED, "?"
PROMPT ptext FOR CHAR yn

IF yn = "y" THEN
    EXECUTE u
END IF
END WHILE
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 that is declared with the FOR UPDATE keywords. For more information on the update cursor, see page 7-111.

References

In this manual, see the following statements: DECLARE, INSERT, OPEN, and SELECT.

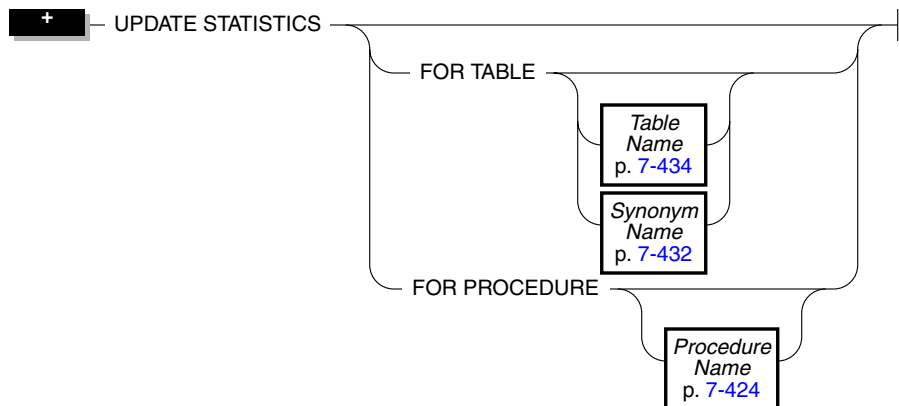
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of the UPDATE statement.

UPDATE STATISTICS

Purpose

Use the UPDATE STATISTICS statement to update the data in the system catalog tables that are used to optimize search strategies and stored procedures.

Syntax



Usage

When you issue an UPDATE STATISTICS statement, IBM Informix OnLine recalculates the data in the **sysables**, **syscolumns**, and **sysindexes** system catalog tables that is used to optimize search strategies. The database server does not update this statistical data automatically. When you issue an UPDATE STATISTICS statement, you trigger the updating process.

Using the UPDATE STATISTICS statement also updates the optimized execution plans for procedures in the **sysproclan** system catalog table. Each time a procedure is executed, the database server optimizes its execution plan if any of the objects referenced in the procedure have changed. It can be useful to optimize a procedure using the UPDATE STATISTICS statement before the procedure is executed, to save time at execution.

The UPDATE STATISTICS statement requires a current database. If you omit the FOR TABLE or FOR PROCEDURE clauses, statistics are updated for every table and procedure in the current database.

If you use the FOR TABLE keywords without a table name, the statistics for all tables in the current database are updated. If you use the FOR PROCEDURE keywords without a procedure name, the statistics for all stored procedures in the current database are updated.

You cannot update the optimizing statistics for a table or procedure that is external to the current database.

When you issue an UPDATE STATISTICS statement, IBM Informix SE recalculates the data in the **systables** system catalog table that is used to optimize search strategies. ♦

When to Update Statistics

Update the system catalog statistics when you perform extensive modifications to a table or 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 causes strong fluctuations in a particular table, you should update the system catalog tables routinely with the UPDATE STATISTICS statement to improve the efficiency of queries.

References

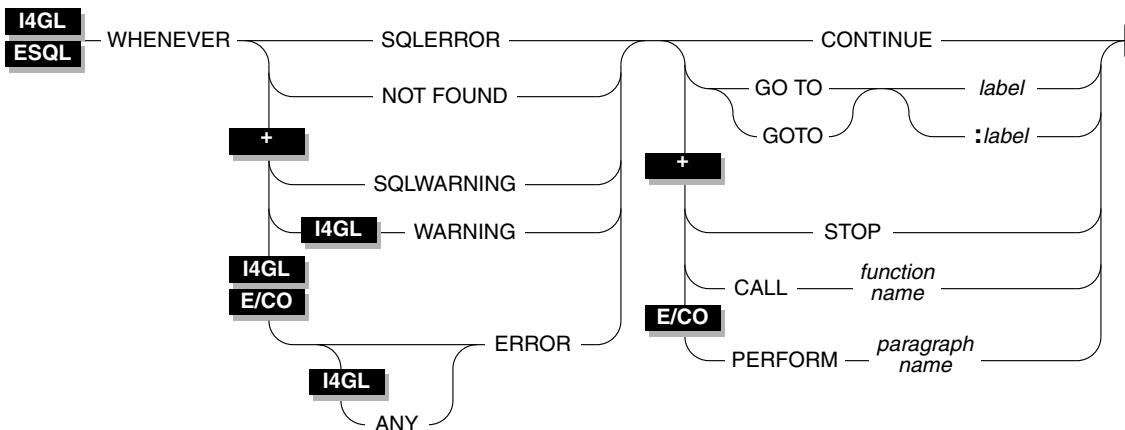
In *IBM Informix Guide to SQL: Tutorial*, see the discussion of UPDATE STATISTICS.

WHENEVER

Purpose

Use the WHENEVER statement to trap errors and warnings that occur during the execution of other SQL statements.

Syntax



function name is the name of an embedded-language function or an IBM Informix 4GL function called when the error or warning condition occurs.

label is a statement label to which program control transfers when the error or warning condition occurs. If the label is an unsigned integer, the statement conforms to ANSI-compliant syntax. If the label is an identifier other than an unsigned integer, the statement is an Informix extension.

paragraph name is the name of a COBOL paragraph.

Usage

Using the `WHENEVER` statement is equivalent to placing an error-checking routine after every SQL statement. If you do not use a `WHENEVER` statement in your program to look for errors or warnings and an error is encountered, execution of the program stops.

The scope of a `WHENEVER` statement is from the location of the statement in the source module until the next `WHENEVER` statement with the same exception condition (`SQLERROR`, `SQLWARNING`, and so on) in the same source module. If there is no other `WHENEVER` statement in the source module, the statement remains in effect until the end of the program or module.

For example, the following `ESQL/C` program has three `WHENEVER` statements, two of which are `WHENEVER SQLERROR` statements. On line 6, the `CONTINUE` keyword is specified; on line 10, `STOP` is used with `SQLERROR`. Any errors encountered after line 6 and before line 10 are ignored. After line 10, and for the rest of the program, any SQL errors encountered cause the program to terminate.

Figure 7-141
ESQL/C program that resets `WHENEVER SQLERROR`

```

1  main()
2  {
3  long char_num;
4
5  $ DATABASE test;
6  $ WHENEVER SQLERROR CONTINUE;
7  printf("\n\nGoing to try first insert\n\n");
8  $ INSERT INTO test_color VALUES ("green");
9  $ WHENEVER NOT FOUND CONTINUE;
10 $ WHENEVER SQLERROR STOP;
11 printf("\n\nGoing to try second insert\n\n");
12 $ INSERT INTO test_color VALUES ("blue");
13 $ CLOSE DATABASE;
14 printf("\n\nProgram over\n\n");
15 }
```


SQLERROR Keyword

If you use the SQLERROR keyword, any SQL statement that fails is handled as directed by the WHENEVER statement. An error occurs whenever the **sqlcode** variable is less than zero. The specification for the **sqlcode** variable for each product is listed in the following table:

4GL	ESQL/C	ESQL/COBOL
SQLCA.SQLCODE STATUS	sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA

The following statement causes SQL errors to be ignored each time they are encountered:

```
WHENEVER SQLERROR CONTINUE
```

If you do not use any WHENEVER SQLERROR statements in a program and if, at compile time, the database accessed by the program is not ANSI-compliant, the default for WHENEVER SQLERROR is STOP.

ANSI

If you do not use any WHENEVER SQLERROR statements in a program and if, at compile time, the database accessed by the program is ANSI-compliant, the default for WHENEVER SQLERROR is CONTINUE. ♦

I4GL

In addition to checking for errors after SQL statements, WHENEVER SQLERROR also checks for errors after Screen I/O statements and VALIDATE statements. ♦

ANY Option

I4GL

ANY Option

If you use the ANY keyword, the **status** variable is set after evaluating an expression, even if it is outside of an SQL statement, Screen I/O statement, or VALIDATE statement. ♦

SQLWARNING Keyword

If you use the SQLWARNING keyword, any SQL statement that generates a warning causes the action indicated by the WHENEVER SQLWARNING statement to be executed. If a warning occurs, the first field of the SQLAWARN record is set to W.

The following statement causes a program to halt execution whenever a warning condition exists:

```
WHENEVER SQLWARNING STOP
```

NOT FOUND Keywords

If you use the NOT FOUND keywords, SELECT and FETCH statements are treated differently from other SQL statements. The NOT FOUND keywords check for the following cases:

- A FETCH statement that attempts to get a row beyond the first or last row in the active set
- A SELECT statement that returns no rows

In both of these cases, the **sqlcode** variable is set to 100. See the figure in [“SQLERROR Keyword” on page 7-339](#) for the name of the **sqlcode** variable in each IBM Informix product.

The following statement calls the **no_rows** function whenever the NOT FOUND condition exists:

```
WHENEVER NOT FOUND CALL no_rows
```

I4GL

Although both NOT FOUND and NOTFOUND indicate the same condition, they cannot be used interchangeably. Use NOTFOUND (one word) in status variables, and use NOT FOUND (two words) with the WHENEVER statement. ♦

WARNING Keyword

I4GL

WARNING is a synonym for SQLWARNING. ♦

I4GL

E/CO

I4GL

I4GL

ERROR Keyword

ERROR is a synonym for SQLERROR. ♦

GOTO Keywords

Use the GOTO clause to transfer control to the statement identified by the label. The keywords GO TO are a synonym for GOTO.

The label specified after the GOTO keyword must be in the same FUNCTION, REPORT, or MAIN program block as the WHENEVER statement. ♦

For example, the WHENEVER statement in the following IBM Informix 4GL code transfers control to the statement labeled **missing**: whenever the NOT FOUND condition occurs:

```
FUNCTION query_data()
...
  FETCH FIRST a_curs INTO p_customer.*
  WHENEVER NOT FOUND GO TO :missing
...
  LABEL missing:
    MESSAGE "No customers found."
    SLEEP 3
    MESSAGE ""
END FUNCTION
```

If your module contains more than one program block, you might need to redefine the error condition. For example, assume the module contains three functions, and the first function includes a WHENEVER...GOTO statement and a corresponding LABEL statement. When compilation moves from the first function to the following function, the error condition still refers to the label; however, the label is no longer defined. If the compiler reads an SQL statement and you have not redefined the error condition (for example, to WHENEVER ERROR CONTINUE), a compilation error results.

You can either reset the error condition by issuing another WHENEVER statement, you can put a labeled statement with the same label-name in each function, or you can use the CALL clause to call a separate function. ♦

If your program contains more than one function, you might need to redefine the error condition. For example, assume the module contains three functions, and the first function includes a `WHENEVER...GOTO` statement and a corresponding labeled statement. When compilation moves from the first function to the following function, the error condition still refers to the label; however, the label is no longer defined. If the compiler reads an SQL statement and you have not redefined the error condition (for example, to `WHENEVER SQLERROR CONTINUE`), a compilation error results.

You either can reset the error condition by issuing another `WHENEVER` statement, you can put a labeled statement with the same label-name in each function, or you can use the `CALL` clause to call a separate function. ♦

CALL Clause

Use the `CALL` clause to transfer program control to the named function. Do not include parentheses after the function name. You cannot pass variables to the function.

The following statement executes a function called `error_recovery` if the program detects an error condition:

```
WHENEVER SQLERROR CALL error_recovery
```

You cannot specify the name of a stored procedure with the `CALL` keyword. If you want to call a stored procedure, use the `CALL` clause to execute a function that contains the `EXECUTE PROCEDURE` statement.

CONTINUE Keyword

Use the `CONTINUE` keyword to instruct the program to take no action. You can use this keyword to turn off a previously specified option.

STOP Keyword

Use the `STOP` keyword to exit from the program immediately. The following statement terminates program execution when the database server issues a warning:

```
WHENEVER SQLWARNING STOP
```

References

In this manual, see the following statements: EXECUTE PROCEDURE and FETCH.

In the *IBM Informix 4GL Reference Manual*, see the following statements: CALL, DEFER, FOREACH, GOTO, IF, and LABEL.

In the user manual for your embedded-language product, see the chapter on error-checking.

Segments

Segments are the elements of syntax that are extracted from the syntax diagrams and discussed separately for better clarification and ease of use.

The following segments, which are common to more than one statement, are gathered in the following section:

- Condition
- Constraint Name
- Database Name
- Data Type
- DATETIME Field Qualifier
- Expression
- Identifier
- Index Name
- INTERVAL Field Qualifier
- Literal DATETIME
- Literal INTERVAL
- Literal Number
- Procedure Name
- Quoted String
- Relational Operator
- Synonym Name
- Table Name
- View Name

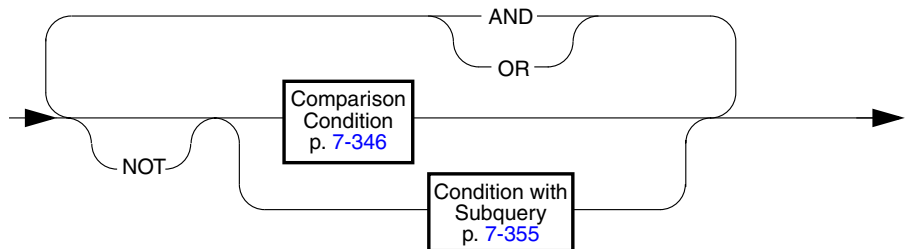
Condition

Purpose

Use a Condition segment to test data to determine whether it meets certain qualifications. You can use the Condition segment in the following ways:

- In an ALTER TABLE or CREATE TABLE statement as a check constraint
- In a DELETE statement within the WHERE clause
- In a SELECT statement within the WHERE clause and the HAVING clause
- In an UPDATE statement within the WHERE clause
- In an IF statement, if you are using SPL
- In a WHILE statement, if you are using SPL

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 only can contain 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.

NOT Operator Option

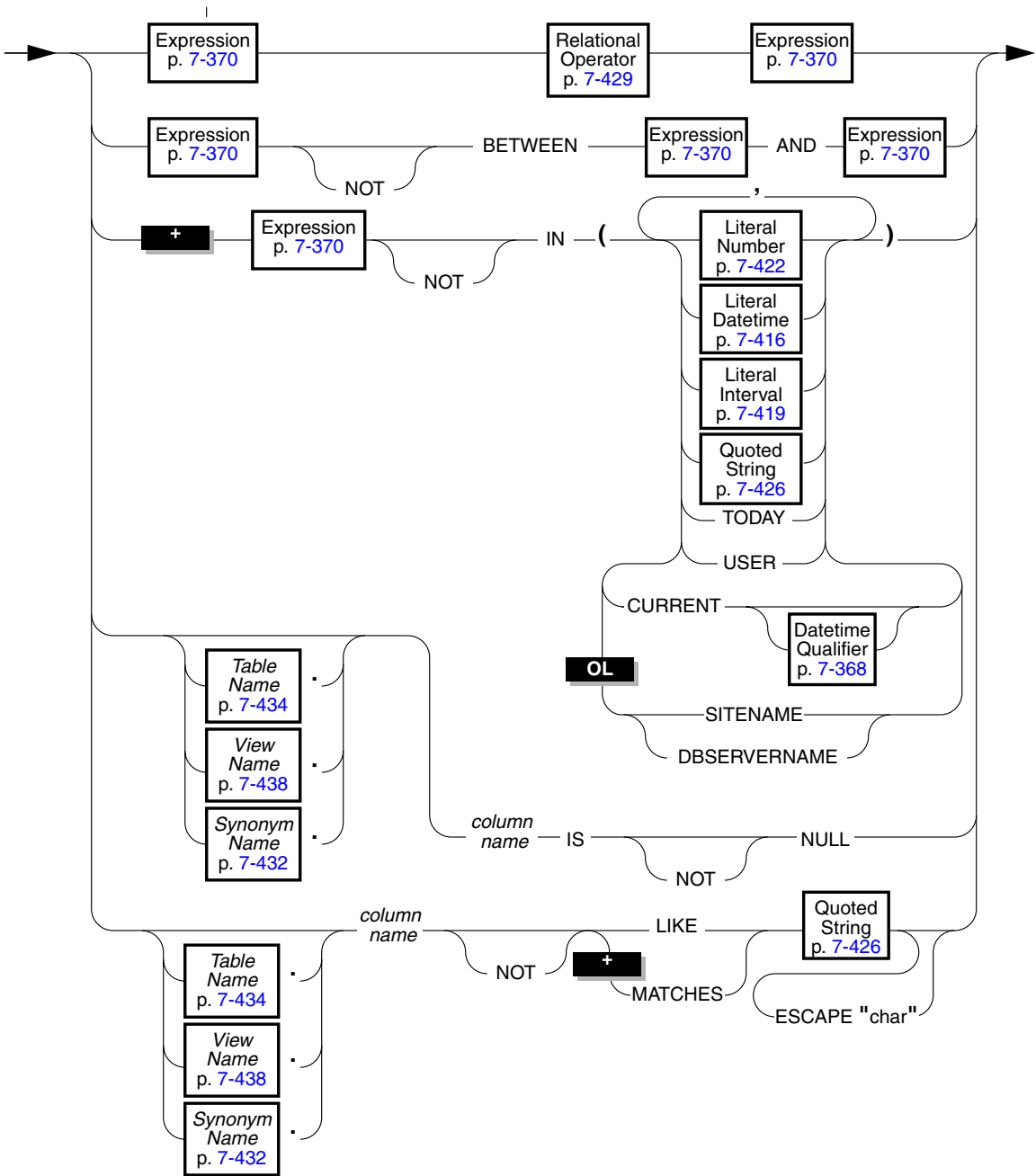
If you preface a condition with the keyword NOT, the test is true only if the condition that the NOT qualifies is false. If the condition qualified by the NOT is unknown (uses a null in the determination), the NOT operator has no effect. The following truth table shows the effect of NOT. The letter T represents a true condition, F represents a false condition, and ? 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)

There are five kinds of comparison conditions: Relational Operator, BETWEEN, IN, IS NULL, and LIKE and MATCHES. Comparison conditions are often called Boolean expressions because they evaluate to a simple true or false result. Their syntax is summarized in the following diagram and explained in detail after the diagram.

Comparison Conditions (Boolean Expressions)



Relational-Operator Condition



Some examples of relational-operator conditions follow.

Figure 7-61
Examples of relational-operator conditions

```
city[1,3] = "San"  
o.order_date > "6/12/86"  
WEEKDAY(paid_date) = WEEKDAY(CURRENT-31 UNITS day)  
YEAR(ship_date) < YEAR (TODAY)  
quantity <= 3  
customer_num <> 105  
customer_num != 105
```

If either expression is null for a row, the condition evaluates to false. For example, if **paid_date** has a null value, you cannot use either of the following statements to retrieve that row.

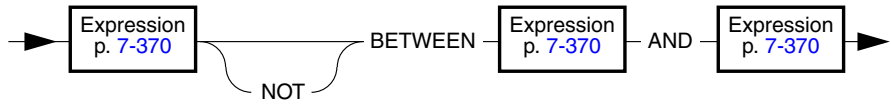
Figure 7-62
Examples of relational-operator conditions that evaluate to false for null values

```
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 an upcoming section.

Figure 7-63
Example of an IS NULL condition

```
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 examples of BETWEEN conditions follow.

Figure 7-64
Examples of BETWEEN conditions

```
order_date BETWEEN "6/1/90" and "9/7/90"

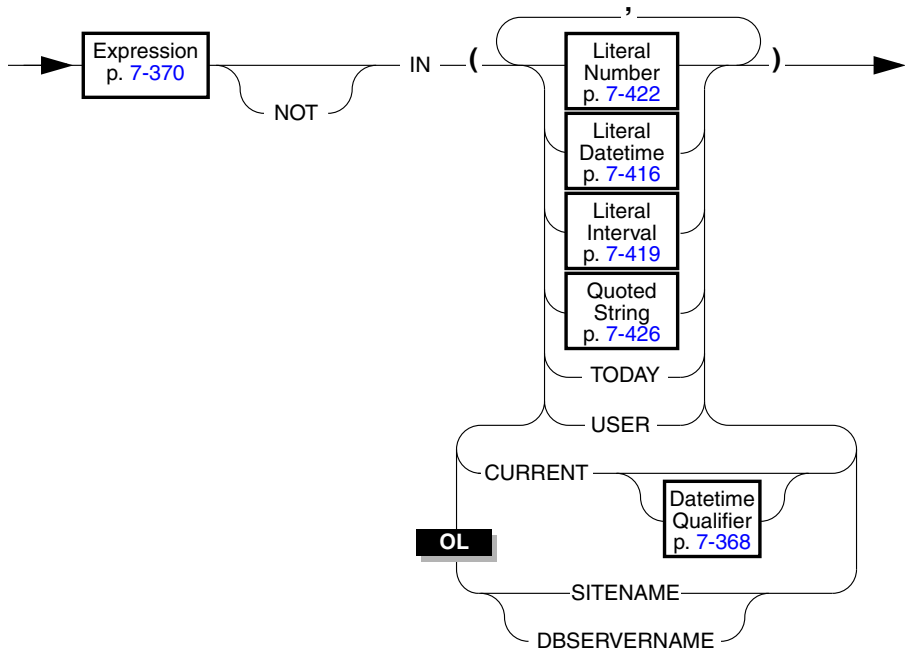
zipcode NOT BETWEEN "94100" and "94199"

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.

Some examples of IN conditions follow.

```
WHERE state IN ("CA", "WA", "OR")
WHERE manu_code IN ("HRO", "HSK")
WHERE user_id NOT IN (USER)
WHERE order_date NOT IN (TODAY)
```

Figure 7-65
Examples of IN conditions

TODAY is evaluated at execution time; CURRENT is evaluated when a cursor is opened, or when the query is executed, if it is a singleton SELECT. ♦

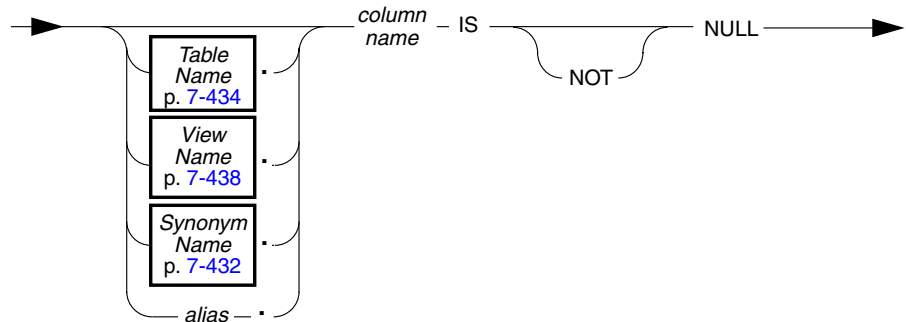
I4GL

ESQL

If you use the USER function, note that it is sensitive to case; it perceives **minnie** and **Minnie** as different values.

IS NULL Condition

The IS NULL condition checks for the presence or absence of null values. The syntax is as follows:



alias is an alias for the table that contains the column.

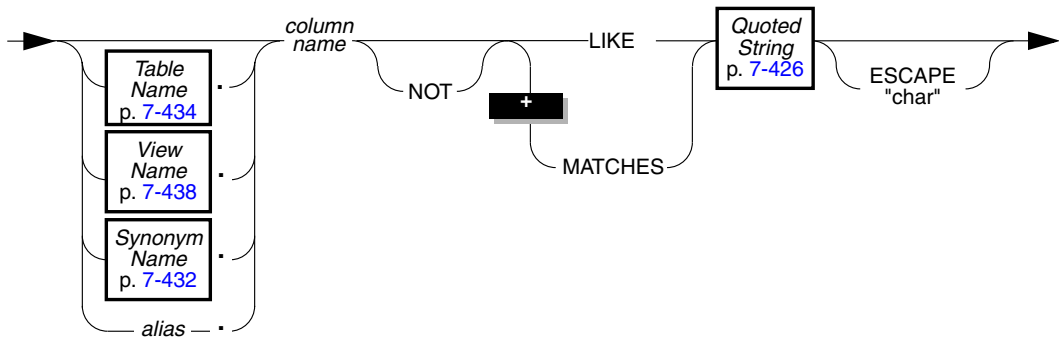
column name is the name of the column to be checked.

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 non-null value. 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 syntax for this test is as follows:



alias is an alias for the table that contains the column.

char is a single character enclosed in quotation marks.

column name is the name of a column.

The search condition is successful when the value of the column on the left matches the pattern specified by the quoted string. You can use wildcard characters in the string. Null values do not satisfy the condition.

You only can use the double quote (") with the quoted string to match a literal quote; you cannot use the ESCAPE keyword. You can use the quote character as the ESCAPE character in matching any other pattern if you write it as "" "".

NOT Option

The NOT option makes the search condition successful when the column on the left has a non-null value and does not match the pattern specified by the quoted string. 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:

- `%` A percent sign matches zero or more characters.
- `_` An underscore matches any single character.
- `\` A backslash removes the special significance of the next character (used to match `%` or `_` by writing `\%` or `_`).

Use of the backslash as an escape character is an Informix extension to ANSI-compliant SQL.

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 since 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:

- `*` An asterisk matches zero or more characters.
- `?` A question mark matches any single character.
- `[...]` Characters within brackets match any of the enclosed characters, including character ranges as in `[a-z]`. A caret (`^`) as the first character within the brackets matches any character that is not listed. Hence `[^abc]` matches any character that is not a, b, or c.
- `\` A backslash removes the special significance of the next character (used to match `*` or `?` by writing `*` or `\?`).

Comparison Conditions (Boolean Expressions)

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 an “F” or “f.”

```
WHERE fname MATCHES "[Ff]*"
```

ESCAPE with LIKE

The ESCAPE clause allows you to 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%` stand for the character `%`. 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"
```

ESCAPE with MATCHES

The ESCAPE clause allows you to include a question mark (`?`), an asterisk (`*`), and a left or right bracket (`[]`) in the quoted string and avoid having them be interpreted as wildcards. If you choose to use `z` as the escape character, the characters `z?` in a string stand for the character `?`. Similarly, the characters `z*` stand for the character `*`. 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 question mark character.

```
SELECT * FROM customer
WHERE company LIKE "*z?*" ESCAPE "z"
```


Condition with a Subquery

You can use a SELECT statement within a condition; this 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 any rows are selected by another SELECT statement

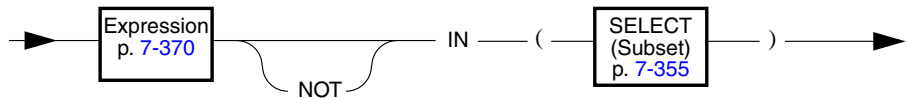
The subquery can depend on the current row being evaluated by the outer SELECT statement; in this case, the subquery is a *correlated subquery*.

There are three kinds of subquery conditions. Each is shown here with its syntax and examples.

Subset of a SELECT Allowed in a Subquery

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 [7-258](#).

IN Subquery



An IN subquery condition is true if the value of the expression matches one or more of the values selected by the subquery. The subquery must return only one column; however, 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 the ALL/ANY/SOME section on page [7-357](#).

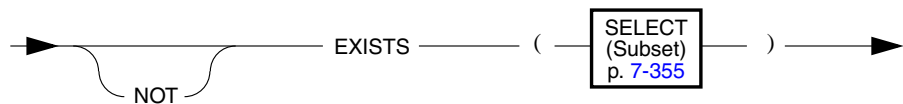
The following condition finds the order numbers for orders that do not include baseball gloves (**stock_num = 1**).

Figure 7-66
Example of an IN subquery

```
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, putting the UNIQUE or DISTINCT keyword into the subquery has no effect on the query results, although eliminating the testing of duplicates can reduce the time needed for running the query.

EXISTS Subquery



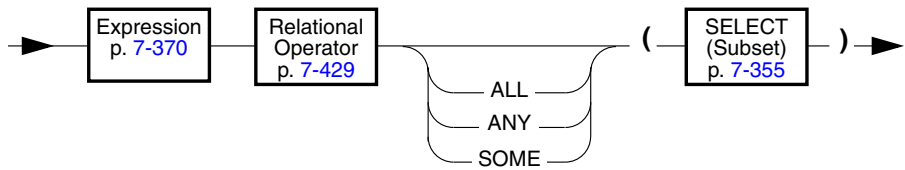
An EXISTS subquery condition evaluates to true if the subquery returns a row. With an EXISTS subquery, one or more columns can be returned. The subquery always contains a reference to a column of the table in the main query. If you use an aggregate function in an EXISTS subquery, at least one row is 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). It is appropriate to use an EXISTS subquery in this SELECT statement because you use the subquery to test both **stock_num** and **manu_code** in **items**.

Figure 7-67
Example of an EXISTS subquery

```
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 a SELECT * in the subquery in place of the column names, since the existence of the whole row is tested, not specific column values.

ALL/ANY/SOME Subquery

ALL is a keyword that denotes that the search condition is true if the comparison is true for every value returned by the subquery. If the subquery returns no value, the condition is true.

ANY is a keyword that denotes that the search condition is true if the comparison is true for at least one of the values returned. If the subquery returns no value, the search condition is false.

SOME is an alias for **ANY**.

In the following example, the first condition tests whether each **total_price** is greater than the total price of every item in order number 1023. The second condition produces the same results by using the **MAX** aggregate function.

Figure 7-68

Example of an ALL subquery and an equivalent aggregate subquery

```
total_price > ALL (SELECT total_price FROM items
                   WHERE order_num = 1023)

total_price > (SELECT MAX(total_price) FROM items
               WHERE order_num = 1023)
```

The following conditions are true when the total price is greater than the total price of at least one of the items in order number 1023. The first condition uses the **ANY** keyword; the second uses the **MIN** aggregate function.

Figure 7-69

Example of an ANY subquery and an equivalent aggregate subquery

```
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 of the subquery values. For example, the following condition is true when the expression **total_price** is not greater than any of the selected values. That is, it is true when **total_price** is greater than none of the total prices in order number 1023.

Figure 7-70

Example of the keyword NOT with an ANY subquery

```
NOT total_price > ANY (SELECT total_price FROM items
                       WHERE order_num = 1023)
```

Omitting ANY, ALL, or SOME Keywords

You can omit the keywords ANY, ALL, or SOME in a subquery if you know that the subquery will return exactly one value. If you omit the ANY, ALL, or SOME keywords and the subquery returns more than one value, you receive an error. The subquery in the following example returns only one row because it uses an aggregate function:

```
SELECT order_num FROM items
       WHERE stock_num = 9 AND quantity =
             (SELECT MAX(quantity) FROM items WHERE stock_num = 9)
```

Conditions with AND or OR

You can combine simple conditions with the logical operators AND or OR. The following SELECT statements contain examples of complex conditions in their WHERE clauses.

Figure 7-71

Combining simple conditions with AND or OR

```
SELECT customer_num, order_date FROM orders
       WHERE paid_date > "1/1/90" 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 ? 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 condition 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. Since **ship_weight** is null, **ship_charge/ship_weight** is also null; therefore, the truth value of **ship_charge/ship_weight** < 5 is UNKNOWN. Since **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.

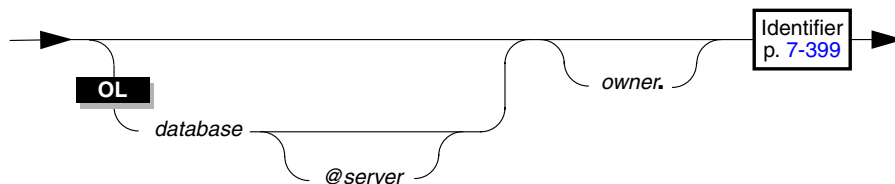
Constraint Name

Purpose

Use the Constraint Name syntax wherever you see a reference to a constraint name in a syntax diagram. The Constraint Name segment appears in the following statements:

- ALTER TABLE
- CREATE TABLE
- SET CONSTRAINTS

Syntax



database is the name of the database in which the constraint resides.

owner is the user name of the owner of the constraint. If you are using an ANSI-compliant database, you must use the *owner.* convention for a constraint that you do not own.

server is the name of the IBM Informix OnLine database server that is home to *database*. The at sign (@) is a literal character that you must use to introduce the database server name.

Usage

The actual name of the constraint is an SQL identifier.

If you are creating a table, the *.name* of the constraint must be unique within a database.

ANSI

If you are creating a table, the combination *owner.name* must be unique within a database.

The *owner.name* is case sensitive. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page [7-436](#). ♦

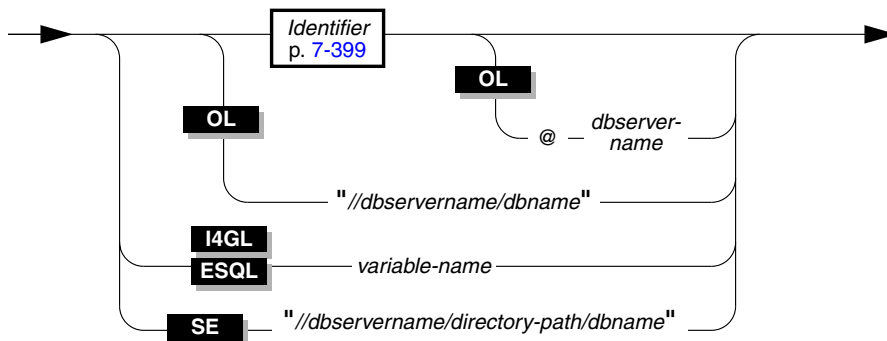
Database Name

Purpose

Use the Database Name syntax wherever you see a reference to a database name in a syntax diagram. The Database Name segment is used in the following statements:

- CREATE DATABASE
- DATABASE
- DROP DATABASE
- ROLLFORWARD DATABASE
- START DATABASE

Syntax



dbname is the name of the database itself.

dbservername is the name of the database server that is home to the database.

directory-path is the path of the database directory up to the parent directory of the **.dbs** directory.

variable-name is a program or host variable that contains the name of a database.

Usage

The simple database name is an SQL identifier, as described on page 7-399. If you are creating a database, the name that you assign to the database can be up to 18 characters long. Database names are not case sensitive.

The following example shows a database specification:

```
empinfo@personnel
```

@dbservername Option

If you use a database server name, do not put any spaces between the names and the @. For example, the following statement is valid for the **stores5** database on the **training** database server:

```
DATABASE stores5@training
```

Specifying a database server name allows you to choose a database on another database server as your current database. You can name the current database server using *db-server-name*, even though that is extra information.

//dbservername/dbname Option

If you use the alternative naming method, do not put spaces between the quotes, slashes, and names, as shown in the following example:

```
DATABASE "//training/stores5"
```

As with the *@dbservername* option, specifying a database server name allows you to choose a database that is on another database server as your current database. You can name the local database server by using *dbservername* along with the *dbname*.

variable-name Option

You can use a variable within an 4GL or embedded-language program to hold the name of a database. ♦

4GL

ESQL

//dbservername/directory-path/dbname Option

I4GL

You can use a variable to hold a database name within a function or the MAIN program block. You cannot use a variable to hold a database name if you are using a DATABASE statement to define global variables defined LIKE database columns. ♦

SE

I4GL

If you want to specify a database that neither resides in your current directory nor in a directory specified by the **DBPATH** environment variable, you must follow the DATABASE keyword with a program variable that evaluates to the full pathname of the database (excluding the **.dbs** extension). ♦

ESQL

SE

INET

//dbservername/directory-path/dbname Option

If you are using IBM Informix NET, you can specify a database on a different database server. Do not put spaces between the quotes, slashes, and names. The following database name describes a **stores5** database that resides on the **business** database server:

```
//business/u/acctng/demo/stores5
```

♦

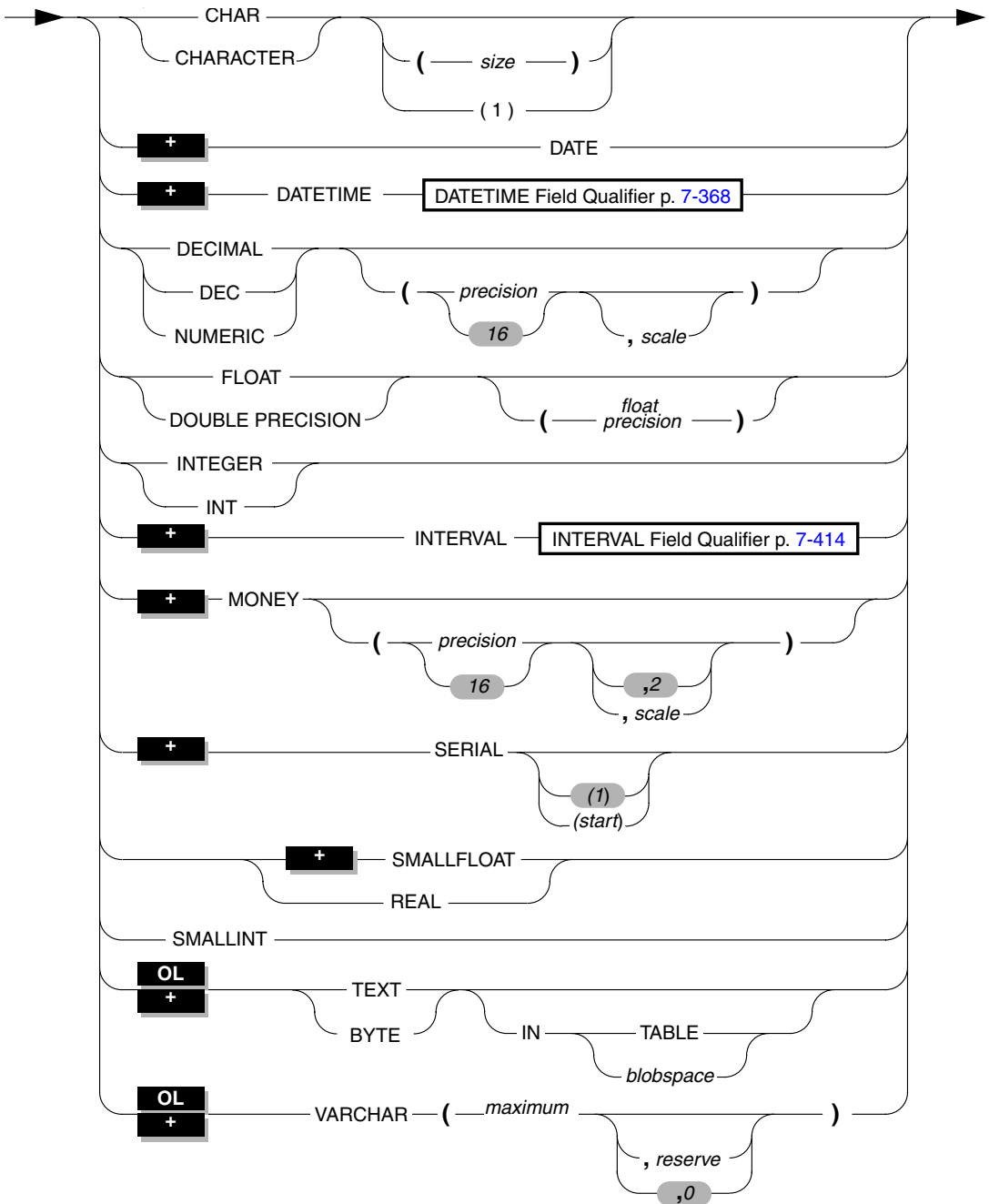
Data Type

Purpose

Use the Data Type segment whenever you have to specify the data type of a column or value. The Data Type segment is used in the following statements:

- ALTER TABLE
- CREATE PROCEDURE
- CREATE TABLE

Syntax



<i>blob space</i>	is the name of a blob space that already exists.
<i>float precision</i>	is an integer between 1 and 14, inclusive. The <i>float precision</i> is ignored.
<i>maximum</i>	is the maximum possible length of a VARCHAR.
<i>precision</i>	is the total number of significant digits in a decimal or money type; it is an integer between 1 and 16, inclusive.
<i>reserve</i>	is the amount of space reserved for a VARCHAR even if the actual data is shorter than <i>reserve</i> .
<i>scale</i>	is the number of digits to the right of the decimal point.
<i>size</i>	is the number of characters in the column.
<i>start</i>	is the starting number for values in a SERIAL column.

Usage

For more information, see the discussion of all the data types in [Chapter 3, "Data Types."](#)

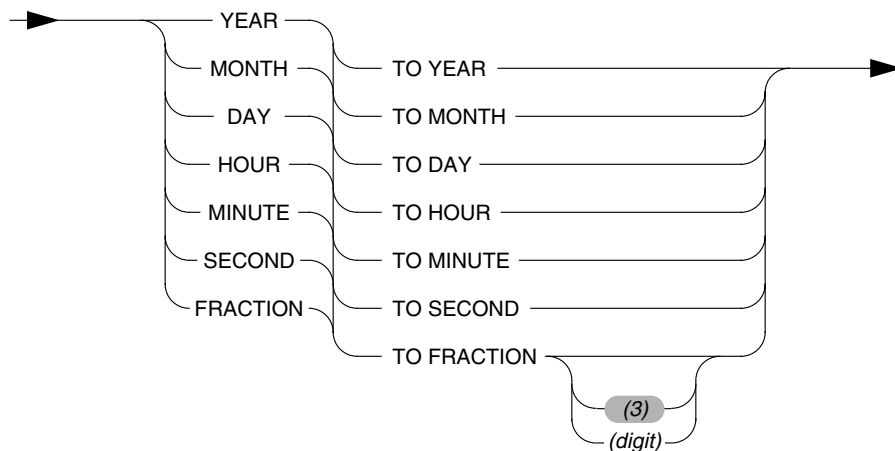
DATETIME Field Qualifier

Purpose

The DATETIME field qualifier specifies the largest and smallest unit of time in a DATETIME column or value. Use the DATETIME field qualifier with the following segments:

- Data type
- Expression (in a constant expression)

Syntax



digit

is a single integer between 1 and 5 indicating to how many digits of precision the fraction is measured.

Usage

Specify the largest unit for the first DATETIME value; after the word TO, specify the smallest unit for the value. The keywords imply that the following values are used in the DATETIME object:

YEAR	A year numbered from A.D. 1 to 9999.
MONTH	A month, numbered from 1 to 12.
DAY	A day, numbered from 1 to 31, as appropriate to the month in question.
HOUR	An hour, numbered from 0 (midnight) to 23.
MINUTE	A minute, numbered from 0 to 59.
SECOND	A second, numbered from 0 to 59.
FRACTION	A fraction of a second, with up to five decimal places. The default scale is three digits (thousandth of a second).

Some examples of DATETIME qualifiers follow.

```
DAY TO MINUTE
YEAR TO MINUTE
DAY TO FRACTION(4)
MONTH TO MONTH
```

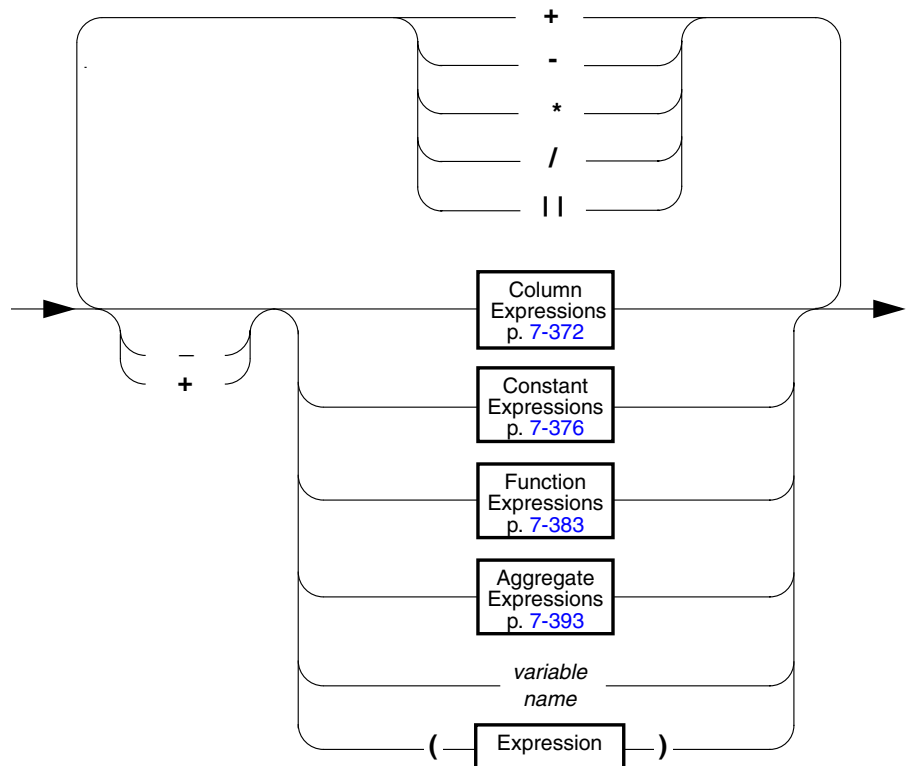
Figure 7-72
Examples of DATETIME qualifiers

Expression

An expression is one or more pieces of data contained in or derived from the database or database server. The Expression segment is used in the following statements and segments:

- SELECT
- DELETE within the Condition segment
- UPDATE within the Condition segment
- In an SPL Expression

Syntax



variable name is the name of a program or host variable that contains a value.

Usage

You can combine expressions by connecting 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 together. For example, the following 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**.

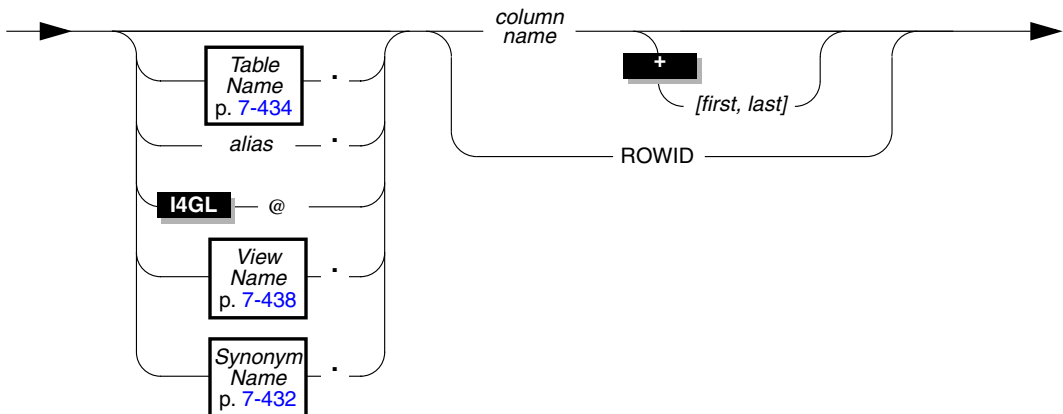
```
lname[1,3] || zipcode  
$file_variable || ".dbg"  
"Date:" || TODAY
```

You cannot use the concatenation operator in an ESQL-only statement. The ESQL-only statements are as follows:

ALLOCATE DESCRIPTOR	FETCH
CLOSE	FLUSH
DEALLOCATE DESCRIPTOR	FREE
DECLARE	GET DESCRIPTOR
DESCRIBE	OPEN
EXECUTE	PREPARE
EXECUTE IMMEDIATE	PUT

Column Expressions

The possible syntax for column expressions is as follows:



alias is used in a SELECT statement in any clause but the SELECT and FROM clauses; it is the alternative name for the table as established in the FROM clause.

column name is the name of the column that you are selecting.

first is the position of the first character of the column (CHAR, VARCHAR, or TEXT).

last is the position of the last character of the portion that you are selecting.

Some examples of column expressions follow.

```
company
items.price
cat_advert [1,15]
```

Figure 7-73
Examples of column expressions

Use a table or alias name whenever it is necessary to distinguish between columns that have the same name but are in different tables. For example, the following SELECT statements use **customer_num** from both the **customer** and **orders** tables so they precede the column names with the table names.

Figure 7-74

Using table and alias names to distinguish between columns that have the same name

```
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, and TEXT columns. The subscripts indicate the starting and ending character positions contained in the expression. For example, if a value in the **lname** column of the **customer** table is Greenburg, then the following expression evaluates to burg:

```
lname[6,9]
```

Using Rowids

You can use the rowid associated with each row of the table as a property of the row. The rowid is essentially a hidden column. It is unique for each row but it is not necessarily sequential.

The rowid is sequential and starts at 1 for each table. ♦

The following examples show possible uses of the ROWID keyword in a SELECT statement.

Figure 7-75

Using the ROWID keyword in a SELECT statement

```
SELECT *, ROWID FROM customer

SELECT fname, ROWID FROM customer
ORDER BY ROWID
```

SE

Using the At Sign

If your program variable has the same name as a column, you must precede the column name with an at sign (@). If you include the table, view, or alias name with the column name, you do not need the @ sign. ♦

The following examples show column expressions preceded by @ signs.

Figure 7-76

Examples of column expressions preceded by @ signs

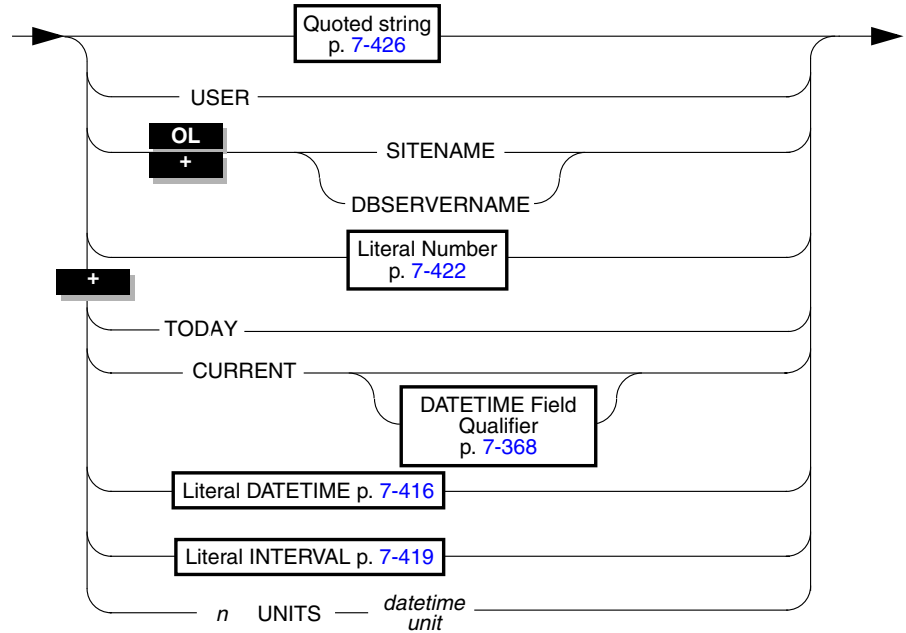
```
SELECT @fname, @lname INTO fname, lname FROM customer
      WHERE customer_num = 119

SELECT stock_num, manu_code FROM stock
      WHERE @stock_num >= stock_num

INSERT INTO customer (@fname, @lname) VALUES (fname,lname)
```

Constant Expressions

The possible syntax for constant expressions is as follows:



datetime unit is one of the units that are used to specify an interval precision; that is, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION.

n is a integer literal number.

Some examples of constant expressions follow.

Figure 7-77
Examples of 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
```

Quoted String as Expression

Some examples of quoted strings as expressions follow.

Figure 7-78
Examples of quoted strings as expressions

```
SELECT "The first name is ", fname FROM customer
INSERT INTO manufact VALUES ("SPS", "SuperSport")
UPDATE cust_calls SET res_dtime = "1990-1-1 10:45"
WHERE customer_num = 120 AND call_code = "B"
```

USER Function

The USER function returns a string containing 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.

Figure 7-79
Examples of 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
```

ANSI

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 of type CHARACTER or VARCHAR and it must be at least eight characters long.

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 are returned by the USER function, with respect to case. ♦

SITENAME and DBSERVERNAME Functions

The SITENAME and DBSERVERNAME functions return the database server name, as defined in the **tbconfig** file for the IBM Informix OnLine installation on which the current database resides. The two function names 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 of type CHARACTER or VARCHAR and must be at least 18 characters long.

In the following example, the first statement returns the name of the database server on which the **customer** table resides. Since 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 only 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.

Figure 7-80
Examples of DBSERVERNAME as an expression

```
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 Expression

Some examples of literal numbers as expressions follow.

Figure 7-81

Examples of literal numbers as expressions

```
INSERT INTO items VALUES (4, 35, 52, "HRO", 12, 4.00)

INSERT INTO acreage VALUES (4, 5.2e4)

SELECT unit_price + 5 FROM stock

SELECT -1 * balance FROM accounts
```

TODAY Function

Use the TODAY function to return the system date as a DATE type. If you specify TODAY as a default value for a column, that column must be of type DATE.

The following statements show how you might use the TODAY function in an INSERT, UPDATE, or SELECT statement.

Figure 7-82

Examples of the TODAY function as an expression

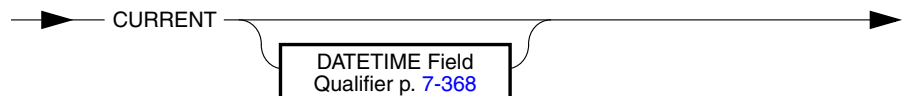
```
UPDATE orders (order_date) SET order_date = TODAY
WHERE order_num = 1005

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 of 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 (page 7-416). If you specify CURRENT as the default value for a column, that column must be of type DATETIME and the qualifier of CURRENT must match the qualifier of the column. 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 value returned is taken from the system clock.

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 sub-second 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 1990 to the current instant.

Figure 7-83

Using the CURRENT function as an expression

```
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 "1990-1-1 00:00:00" AND CURRENT
```

Literal DATETIME as an Expression

Some examples of literal DATETIME as an expression follow.

Figure 7-84*Using literal DATETIME as an expression*

```

SELECT DATETIME (1991-12-6) YEAR TO DAY FROM customer

UPDATE cust_calls SET res_dtime = DATETIME (1990-07-07 10:40)
      YEAR TO MINUTE
WHERE customer_num = 110
AND call_dtime = DATETIME (1990-07-07 10:24) YEAR TO MINUTE

SELECT * FROM cust_calls
WHERE call_dtime
      = DATETIME (1995-12-25 00:00:00) YEAR TO SECOND

```

Literal INTERVAL as an Expression

Some examples of literal INTERVAL as an expression follow.

Figure 7-85*Using 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. Use the following syntax with the UNITS keyword:



datetime unit is one of the units that are used to specify an interval precision; that is, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION (if the unit is YEAR or the expression is a year-month interval; otherwise, it is a day-time interval).

n is a literal number comparable to the datetime unit that you choose (two digits with DAY, for example).

If *n* is not an integer, it is rounded down to the nearest whole number when it is used. The value of *n* must be appropriate for the datetime unit selected. For example, adding one month to "2001-10-31 00:00" results in November 31. Since November has only 30 days, the value of *n* is not appropriate, and an error will be returned.

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.

Figure 7-86

Using the UNITS keyword as an expression

```

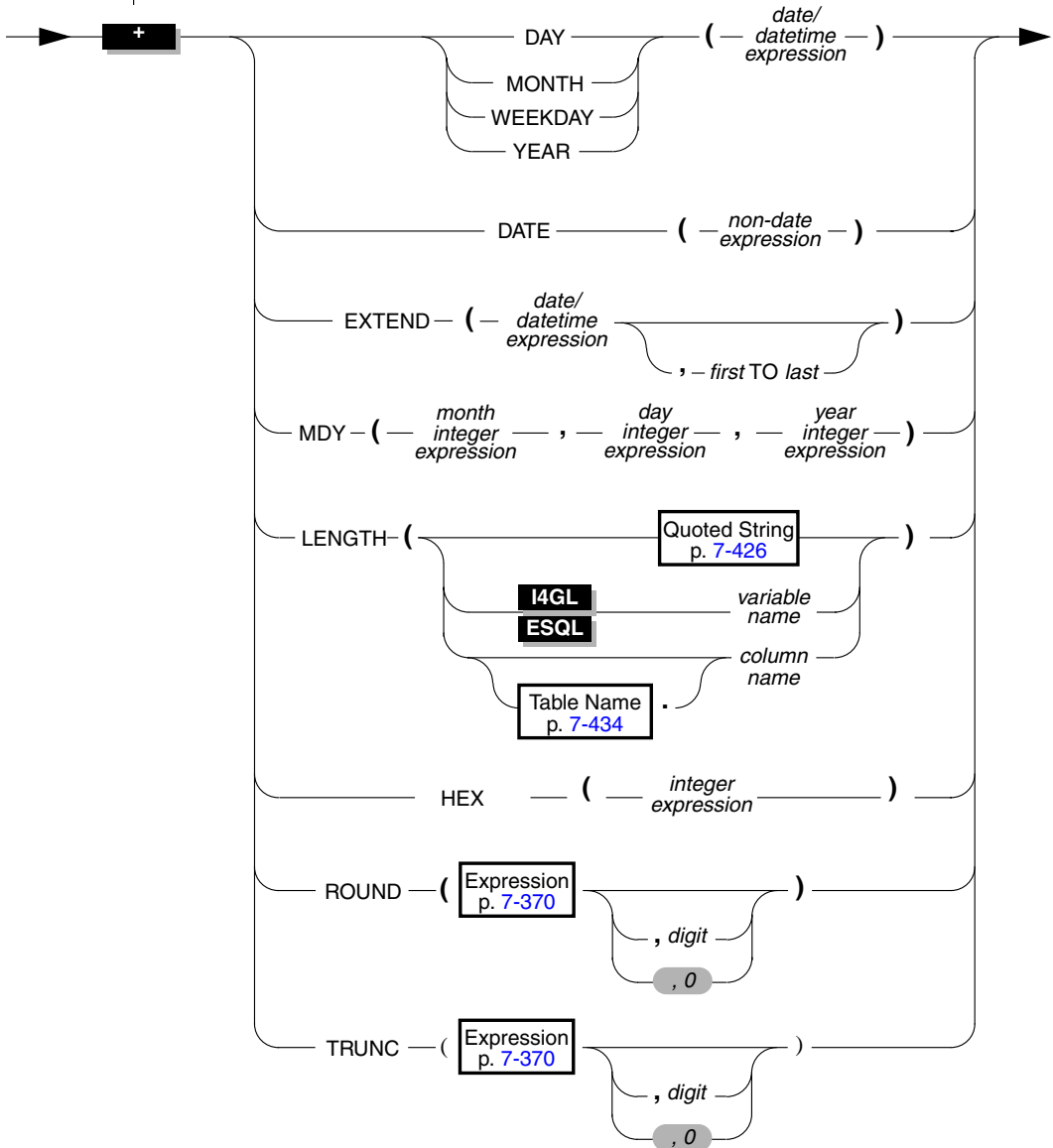
SELECT lead_time + 5 UNITS DAY FROM manufact

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. The syntax for function expressions is as follows:



<i>date/datetime expression</i>	is an expression, as defined on page 7-370 , that evaluates to a DATE or DATETIME value.
<i>day integer expression</i>	is an expression, as defined on page 7-370 , that evaluates to an integer between 1 and 28, 29, 30, or 31, as appropriate for the month.
<i>digit</i>	is an integer between +32 and -32, inclusive, that indicates the digit to which you want to round the expression.
<i>first</i>	is the large qualifier for the DATETIME value.
<i>integer expression</i>	is an expression, as defined on page 7-370 , that can be converted to an integer.
<i>last</i>	is the small qualifier for the DATETIME value.
<i>month integer expression</i>	is an expression, as defined on page 7-370 , that evaluates to an integer between 1 and 12, as appropriate for the month.
<i>non-date expression</i>	is an expression, as defined on page 7-370 , that evaluates to a CHARACTER, DATETIME, or INTEGER value that can be converted to a DATE data type.
<i>variable name</i>	is a program or host variable, the length of which is to be evaluated.
<i>year integer expression</i>	is an expression, as defined on page 7-370 , that evaluates to a four-digit integer between 1 and 9999.

The function expressions are made up of the following three families of functions.

Time Functions	Length Function	Conversion Functions
DATE()	LENGTH()	HEX()
DAY()		ROUND()
EXTEND()		TRUNC()
MDY()		
MONTH()		
WEEKDAY()		
YEAR()		

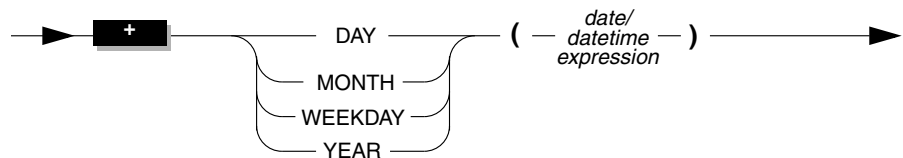
Some examples of function expressions follow.

Figure 7-87
Examples of 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))
```

DAY, MONTH, WEEKDAY, and YEAR Functions

The syntax for these four functions is all the same, as shown here:



date/datetime expression is an expression, as defined on page 7-370, that evaluates to a DATE or DATETIME value.

DAY Function

The DAY function returns an integer that represents the day of the month. For example, the following WHERE clause uses the DAY function with the CURRENT function to compare column values to the current day of the month:

```
WHERE DAY(order_date) > DAY(CURRENT)
```

MONTH Function

The MONTH function returns an integer corresponding to the month portion of its type DATE or DATETIME argument. For example, the following query returns a number (1 through 12) to indicate the month 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. For example, the following query lists all the orders that were paid on the same day of the week as 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 query 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.

DATE Function

The DATE function returns a type DATE value corresponding to the character expression with which you call it. The syntax for the DATE function follows:

→ **+** DATE (*non-date expression*) →

non-date expression is an expression, as defined on page 7-370, that evaluates to a CHARACTER, DATETIME, or INTEGER value that can be converted to a DATE data type.

The argument can be any expression that can be converted to a DATE value, usually a CHAR, DATETIME, or INTEGER value. For example, the following two WHERE clauses achieve the same end: converting a string to a date.

```
WHERE order_date < DATE("12/31/90")
```

```
WHERE order_date < DATE(365)
```

EXTEND Function

The EXTEND function adjusts the precision of a DATETIME or DATE value. Its syntax is as follows:

→ **+** EXTEND (*date/datetime expression* , *first TO last*) →

date/datetime expression is an expression, as defined on page 7-370, that evaluates to a DATE or DATETIME value.

first is a qualifier that specifies the first field in the result. It can be any DATETIME qualifier, as defined on page 7-368, as long as *first* is larger than *last*.

last is a qualifier that specifies the last field in the result.

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 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 examples, the first EXTEND call evaluates to the `call_dtime` column value of YEAR TO SECOND. The second example 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.

Figure 7-88
Examples of the EXTEND function

```
EXTEND (call_dtime, YEAR TO SECOND)

EXTEND (DATETIME (1989-8-1) YEAR TO DAY, YEAR TO MINUTE)
      - INTERVAL (720) MINUTE (3) TO MINUTE
```

MDY Function

The MDY function returns a type DATE value with three expressions that evaluate to integers representing the month, day of the month, and year. The syntax for MDY is as follows:

→ **+** MDY ($\frac{\text{month}}{\text{integer expression}}$, $\frac{\text{day}}{\text{integer expression}}$, $\frac{\text{year}}{\text{integer expression}}$) →

day integer expression is an expression, as defined on page 7-370, that evaluates to an integer between 1 and 28, 29, 30, or 31, as appropriate for the month.

month integer expression is an expression, as defined on page 7-370, that evaluates to an integer between 1 and 12, as appropriate for the month.

year integer expression is an expression, as defined on page 7-370, that evaluates to a four-digit integer.

The first expression must evaluate to an integer representing the number of the month (1 to 12).

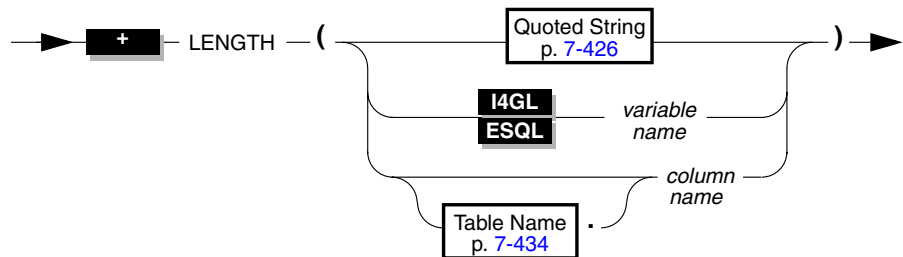
The second expression must evaluate to an integer representing the number of the day of the month (1 to 28, 29, 30, or 31, as appropriate for the month.)

The third expression must evaluate to a four-digit integer representing the year. You cannot use a two-digit abbreviation for the third expression. The following UPDATE statement 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"
```

LENGTH Function

The LENGTH function returns the length of a character column, not including any trailing spaces. With TEXT or BYTE columns, the LENGTH function returns the full number of bytes in the column.



column name is the name of a column.

variable name is a program variable or a host variable that contains a character string.

The LENGTH function allows only one argument. You can, however, combine LENGTH values through an expression, as shown in the following example:

```
SELECT customer_num, LENGTH(fname) + LENGTH(lname)
LENGTH("How many bytes is this?")
FROM customer WHERE LENGTH(company) > 10
```

You can use the LENGTH function to return the length of a character variable. ♦

I4GL

ESQL

HEX Function

The HEX function returns the hexadecimal encoding of an integer expression.

→ _____ HEX _____ (_____ *integer expression* _____) →

integer expression is an expression, as defined on page 7-370, that can be converted to an integer.

The following SELECT statement shows several uses of the HEX function:

```
SELECT HEX(customer_num), HEX(zipcode), HEX(ROWID),
       HEX(LENGTH(fname)) FROM customer
```

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 second-lowest and lowest bits. For VARCHAR columns, you can determine the minimum space and maximum space from the second-lowest and lowest bits. (See “SYSCOLUMNS” on page 2-13 for more information about encoded information.)

```
SELECT colname, HEX(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 bits in the hexadecimal number constitute the dbspace number and are used to identify the table in **tbcheck** output.

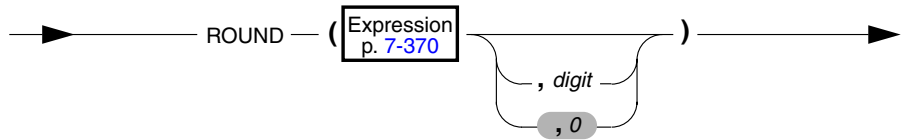
```
SELECT tabname, HEX(partnum) FROM systables
```

You can combine HEX values through an expression, as shown in the following example:

```
SELECT HEX(order_num + 1) FROM orders
```

ROUND Function

The ROUND function returns the rounded value of an expression.



digit is an integer between +32 and -32, inclusive, that indicates the digit to which you want to round the *expression*.

The expression must be numeric or must be convertible to numeric.

If you omit the digit indication, the value is rounded to zero digits or to the ones place. Positive digit values indicate rounding to the right of the decimal point; negative digit values indicate rounding to the left of the decimal point.

	expression: 24536.8746
ROUND (24536.8746, -2) = 24500.00	↓ ↓ ↓
ROUND (24536.8746, 0) = 24537.00	↓ ↓ ↓
ROUND (24536.8746, 2) = 24536.87	-2 0 2

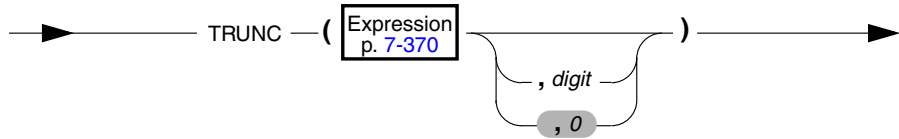
```
SELECT order_num , ROUND(total_price) FROM items
WHERE ROUND(total_price) = 124.00
```

If you use a MONEY data type as the argument for the ROUND function and you round to zero places, the value is displayed with .00. For example, the following SELECT statement 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
```

TRUNC Function

The TRUNC function returns the truncated value of a numeric expression.



digit is an integer between +32 and -32, inclusive, that indicates the digit to which you want to truncate the expression.

The expression must be numeric or of 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 ones place. Positive digit values indicate truncating to the right of the decimal point; negative digit values indicate truncating to the left of the decimal point.

	expression: 24536.8746		
TRUNC (24536.8746, -2) =24500	↓	↓	↓
TRUNC (24536.8746, 0) = 24536	↓	↓	↓
TRUNC (24536.8746, 2) = 24536.87	-2	0	2

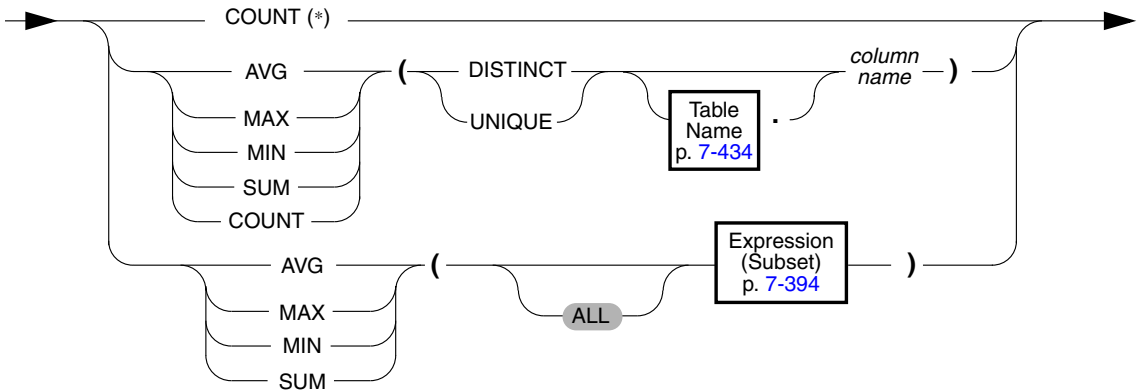
If you use a MONEY data type as the argument for the TRUNC function and you truncate to zero places, the .00 places are removed. For example, the following SELECT statement truncates a MONEY value and an INTEGER value. It displays 125 and a truncated price in the form xxx for each row in **items**.

```
SELECT TRUNC(125.46), TRUNC(total_price) FROM items
```

Aggregate Expressions

An aggregate expression uses an aggregate function to summarize selected database data.

The syntax of aggregate function expressions is as follows:



column name is the name of the column.

first is the large qualifier for the DATETIME value.

last is the small qualifier for the DATETIME value.

An aggregate function returns one value for a set of queried rows. Some examples of aggregate functions in SELECT statements follow.

Figure 7-89

Examples of 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
```

Subset of Expressions Allowed in an Aggregate Expression

The argument of an aggregate function cannot itself contain an aggregate function. For example, you cannot use `MAX(AVG(order_num))`. You cannot use an aggregate function in a `WHERE` clause unless it is contained in a subquery. You cannot use an aggregate function on a `BYTE` or `TEXT` column. For the full syntax of an expression, see page [7-370](#).

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 of the values selected from the named column or expression, including any duplicate values, are used in the calculation.

COUNT(*) Keyword

The `COUNT (*)` keyword returns the number of rows that satisfy the `WHERE` clause. The following query finds how many Hero products are stocked.

```
SELECT COUNT(*) FROM stock WHERE manu_code = "HRO"
```

If the `SELECT` statement contains a `GROUP BY` clause, the `COUNT(*)` keyword reflects the number of values in each group. For example, the following statement is grouped by the first name; the rows are selected if there is more than one occurrence of the same name.

```
SELECT fname, COUNT(*) FROM customer
GROUP BY fname
WHERE 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.

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 over only the distinct values in the specified column or expression. The following query finds the average price of a helmet:

```
SELECT AVG(unit_price) FROM stock WHERE stock_num = 110
```

Because all of the helmets have different unit prices, using the DISTINCT keyword would have no effect in this query.

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 following query 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.

```
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. If you use the DISTINCT keyword, the sum is over 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.

COUNT Keyword

The COUNT keyword returns the number of different values in the column or expression. 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.

Summary of Aggregate Function Behavior

The following table summarizes the action of the aggregate functions. If a query is constructed as follows:

```
SELECT a_number FROM testtable WHERE a_number < 10
```

such that it returns the following values:

```
a_number
2
2
2
3
3
4
(null)
```

then the following results occur for each of the functions listed:

Function	Results
COUNT(*)	7
AVG	2.67
AVG (DISTINCT)	3
MAX	4
MAX(DISTINCT)	4
MIN	2
MIN(DISTINCT)	2
SUM	16
SUM(DISTINCT)	9
COUNT(DISTINCT)	3

For example, the following query returns the value 3:

```
SELECT AVG(DISTINCT a_number) FROM testtable WHERE a_number < 10
```

Error Checking with Aggregate Functions

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 of the 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. ♦

I4GL

ESQL

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.

Figure 7-90
Examples of arithmetic operators

```
quantity * total_price  
price * 2 doubleprice  
COUNT(*) + 2
```

If any value that participates in an arithmetic expression is null, the value of the entire expression is null. For example, consider the following query:

```
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** also is 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.

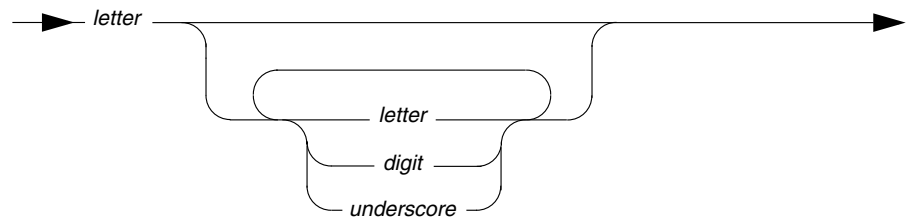
Identifier

Purpose

Use the Identifier segment in the following segments to specify the name of a database object:

- Constraint Name
- Database Name
- Index Name
- Synonym Name
- Table Name
- View Name
- Procedure Name

Syntax



digit is an integer from 0 to 9.

letter is an uppercase or lowercase character from a to Z.

underscore is the underscore (`_`) character.

Usage

An identifier can contain up to 18 characters.

Do not declare as an identifier any reserved word of your application development tool.



Tip: *If you receive an error message that seems unrelated to the statement that caused the error, you should check to determine whether the statement uses a reserved word as an identifier.*

The following list specifies all of the ANSI reserved words. If you use one of these words as an identifier and set **DBANSIWARN** or specify the **-ansi** flag at run time or compile time, you receive a warning.

all	fetch	precision
and	float	primary
any	for	procedure
as	found	privileges
asc	from	public
authorization	go	real
avg	goto	rollback
begin	group	schema
between	having	section
by	in	select
char	indicator	set
character	insert	smallint
check	int	some
close	integer	sql
cobol	into	sqlcode
commit	is	sqlerror
continue	language	sum
count	like	table
create	max	to
current	min	union
cursor	module	unique
dec	not	update
decimal	null	user
declare	numeric	values
delete	of	view
desc	on	whenever
distinct	open	where
double	option	with
end	or	work
escape	order	
exec	pascal	
exists	pli	

Potential Ambiguities and Syntax Errors

Although you now can use almost any word as an SQL identifier, syntactic ambiguities can occur. An ambiguous statement might not produce the desired results. This section outlines some of the potential pitfalls and their workarounds.

Using Functions as Column Names

The first pair of examples shows a workaround for using a function as a column name in a SELECT statement. This applies to the aggregate functions (AVG, COUNT, MAX, MIN, SUM), the LENGTH function, the time functions (DATE, DAY, MDY, MONTH, WEEKDAY, YEAR), and the datetime functions (CURRENT and EXTEND).

This use of **avg** as a column name causes the following statement to fail because the database server interprets **avg** as an aggregate function rather than a column name:

```
SELECT avg FROM mytab -- fails
```

This workaround removes ambiguity by including a table name with the column name:

```
SELECT mytab.avg FROM mytab
```

If you use the keyword TODAY, CURRENT, or USER as a column name, ambiguity can occur, as shown in the following example:

```
CREATE TABLE mytab (user char(10),  
                    CURRENT DATETIME HOUR TO SECOND, TODAY DATE)  
  
INSERT INTO mytab VALUES ("josh", "11:30:30", "1/22/89")  
  
SELECT user, current, today FROM mytab
```

The database server interprets **user**, **current**, and **today** in the SELECT statement as the SQL functions USER, CURRENT, and TODAY. Thus, instead of returning `josh, 11:30:30, 1/22/89`, the SELECT statement returns the current user name, the current time, and the current date.

If you want to select the actual columns of the table, you must write the SELECT statement in one of these two ways:

```
SELECT mytab.user, mytab.current, mytab.today FROM mytab;
```

or

```
$SELECT * FROM mytab;
```

Using Keywords as Column Names

There are specific workarounds 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

The first pair of examples shows a workaround for using the keyword ALL (or DISTINCT or UNIQUE) in a SELECT statement.

This use of **all** as a column name causes the following statement to fail because the database server interprets **all** as a keyword rather than as a column name:

```
SELECT all FROM mytab -- fails
```

This example shows a workaround using the keyword ALL with the column name **all**:

```
SELECT ALL all FROM mytab
```

The following set of examples shows two workarounds for using the keywords UNIQUE or DISTINCT as a column name in a CREATE TABLE statement.

This use of **unique** as a column name causes the following statement to fail because the database server interprets **unique** as a keyword rather than as a column name:

```
CREATE TABLE mytab (unique INTEGER) -- fails
```

The following workaround uses two SQL statements. The first statement creates the column **mycol** and the second renames the column **mycol** to **unique**:

```
CREATE TABLE mytab (mycol INTEGER)

RENAME COLUMN mytab.mycol TO unique
```

The following workaround also uses two SQL statements. The first statement creates the column **mycol** and 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 next set of examples shows two workarounds for using the keyword **INTERVAL** (or **DATETIME**) as a column name in a **SELECT** statement.

This use of **interval** as a column name causes the following statement 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
```

The following workaround removes ambiguity by specifying a table name with the column name:

```
SELECT mytab.interval FROM mytab;
```

Another workaround includes an owner name with the table name:

```
SELECT josh.mytab.interval FROM josh.mytab;
```

Using rowid as a Column Name

Every table has a virtual column named **rowid**. To avoid ambiguity, you cannot use **rowid** as a column name. The following actions cause 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**

The term **rowid** can, however, be used as a table name. For example:

```
CREATE TABLE rowid (column INTEGER,
                    date DATE, char CHAR(20))
```

Using Keywords as Table Names

The following examples show workarounds that involve owner naming when the keyword **STATISTICS** or **OUTER** is used as a table name. This also applies to the use of **STATISTICS** or **OUTER** as a view name or synonym.

This use of **statistics** as a table name causes the following statement 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
```

This example shows a workaround that specifies an owner name with the table name, to avoid ambiguity:

```
UPDATE josh.statistics SET mycol = 10
```

This use of **outer** as a table name causes the following statement to fail because the database server interprets **outer** as a keyword for performing an outer join:

```
SELECT mycol FROM outer -- fails
```

This workaround 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 a number of these situations. You can use the keyword AS to provide a workaround for the exceptions.

You can use the keyword AS in front of column labels or table aliases. The AS keyword is an Informix extension to SQL.

This is the syntax for using the AS keyword with a column label:

```
column-name AS display-label FROM table-name
```

This is the syntax for using the AS keyword with a table alias:

```
SELECT select-list FROM table-name AS table-alias
```

Using AS with Column Labels

The following examples of workarounds use the keyword AS with a column label. The first pair shows how you can use the keyword UNITS (or YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION) as a column label.

This use of **units** as a column label causes the following statement to fail because the database server interprets it as a DATETIME qualifier for the column named **mycol**:

```
SELECT mycol units FROM mytab
```

This statement uses a workaround that includes the keyword AS:

```
$SELECT mycol AS units FROM mytab;
```

The following examples show how the keywords AS or FROM can be used as a column label.

This use of **as** as a column label causes the statement 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
```

This statement uses a workaround that repeats the keyword AS:

```
SELECT mycol AS as from mytab
```

This use of **from** as a column label causes the following statement to fail because the database server expects a table name to follow the first **from**:

```
SELECT mycol from FROM mytab -- fails
```

This workaround uses the keyword AS to identify the first **from** as a column label:

```
SELECT mycol AS from FROM mytab
```

Using AS with Table Aliases

The following examples of workarounds use the keyword AS with a table alias. The first pair shows how to use the keyword ORDER (or FOR, GROUP, HAVING, INTO, UNION, WITH, CREATE, GRANT, or WHERE) as a table alias.

This use of **order** as a table alias causes the following statement to fail because the database server interprets **order** as part of an ORDER BY clause:

```
SELECT * FROM mytab order -- fails
```

This workaround uses the keyword AS to identify **order** as a table alias:

```
SELECT * FROM mytab AS order;
```

The next pair of examples shows how to use the keyword WITH as a table alias.

This use of **with** as a table alias causes the following statement to fail because the database server interprets the keyword as part of the WITH CHECK OPTION syntax:

```
SELECT * FROM mytab with -- fails
```

This workaround uses the keyword AS to identify **with** as a table alias:

```
$SELECT * FROM mytab AS with;
```

The following pair of examples shows how to use the keyword CREATE (or GRANT) as a table alias.

This use of **create** as a table alias causes the following statement 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:

```
$SELECT * FROM mytab create -- fails
```

This workaround uses the keyword AS to identify **create** as a table alias:

```
$SELECT * FROM mytab AS create;
```

Fetching Keywords as Cursor Names

In a few situations, there is no workaround 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 generates a syntax error because the preprocessor interprets the syntax as pertaining to a scroll cursor 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.

```
$DECLARE next CURSOR FOR
    SELECT customer_num, lname FROM customer;

$ OPEN next;

$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

If you use the CURRENT, DATETIME, INTERVAL, or NULL keyword as the name of a procedure and use it in an INSERT statement, the syntax of the statement is confused. There is no workaround.

For example, if you define a variable called **datetime**, when you try to insert the value in datetime into a column, you receive a syntax error. This is illustrated 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.

Figure 7-91*Ambiguous use of select and null as variable names*

```
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 make sure it is not the first element in the list. For example, the following example use a workaround to correct the IF statement of [Figure 7-91](#).

```
IF x IN (10, select, 12) THEN LET y = 1; -- problem if
```

There is no workaround 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 is not traced. Instead, the TRACE ON, TRACE OFF, or TRACE PROCEDURE statements are executed. You can trace the value of the variable by making the variable into a more complex expression. [Figure 7-92](#) shows the ambiguous syntax and the workaround.

Figure 7-92

Ambiguous and clear uses of on, off, and procedure with TRACE

```
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 [Figure 7-92](#) conflicts with the syntax for defining global variables. There is no workaround.

Figure 7-93

Ambiguous use of global

```
DEFINE global INT; -- fails;
```


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 for the FOREACH statement. There is no workaround.

For example, the following statements 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. For example, the SELECT statement in the first WHILE statement in the following example is interpreted as a call to the procedure **var1**. The second WHILE statement is interpreted correctly.

```
DEFINE var1, var2 INT;
WHILE var2 = var1
  (SELECT col1 INTO var3 FROM TAB -- error, seen as call var1()
  UNION
  SELECT co2 FROM tab2;
END WHILE

WHILE var2 = var1
  BEGIN
    (SELECT col1 INTO var3 FROM TAB -- ok syntax
    UNION
    SELECT co2 FROM tab2;
  END
END WHILE
```

The SET Keyword in the ON EXCEPTION Statement

If you use a statement that begins with the keyword SET inside the statement ON EXCEPTION, you must enclose it in a BEGIN...END block. This includes the following statements:

```
SET CONSTRAINTS      SET LOCK MODE
SET DEBUG FILE       SET LOG
SET EXPLAIN          SET OPTIMIZATION
SET ISOLATION
```

```
ON EXCEPTION IN (-107)
  SET LOCK MODE TO WAIT; -- error, value expected, not "lock"
END EXCEPTION
```

```
ON EXCEPTION IN (-107)
  BEGIN
  SET LOCK MODE TO WAIT; -- ok
  END
END EXCEPTION
```

References

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of owner naming.

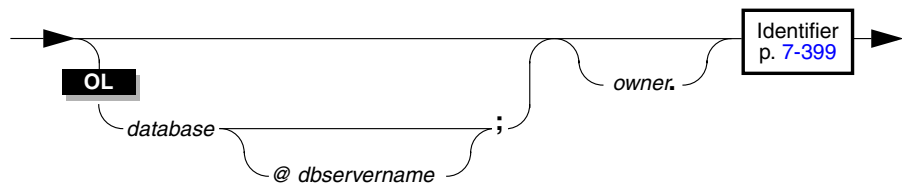
Index Name

Purpose

Use the Index Name segment wherever you see a reference to an index name in a syntax drawing. It appears in the following statements:

- ALTER INDEX
- CREATE INDEX
- DROP INDEX

Syntax



database is the name of the database in which the index resides.

dbservername is the name of the IBM Informix OnLine database server that is home to *database*. The @ sign is a literal character that you must use to introduce the database server name.

owner is the user name of the owner of the index. If you are using an ANSI-compliant database, you must use the *owner.* convention for indexes that you do not own.

Usage

The actual name of the index is an SQL identifier.

If you are creating an index, the *name* must be unique within a database.

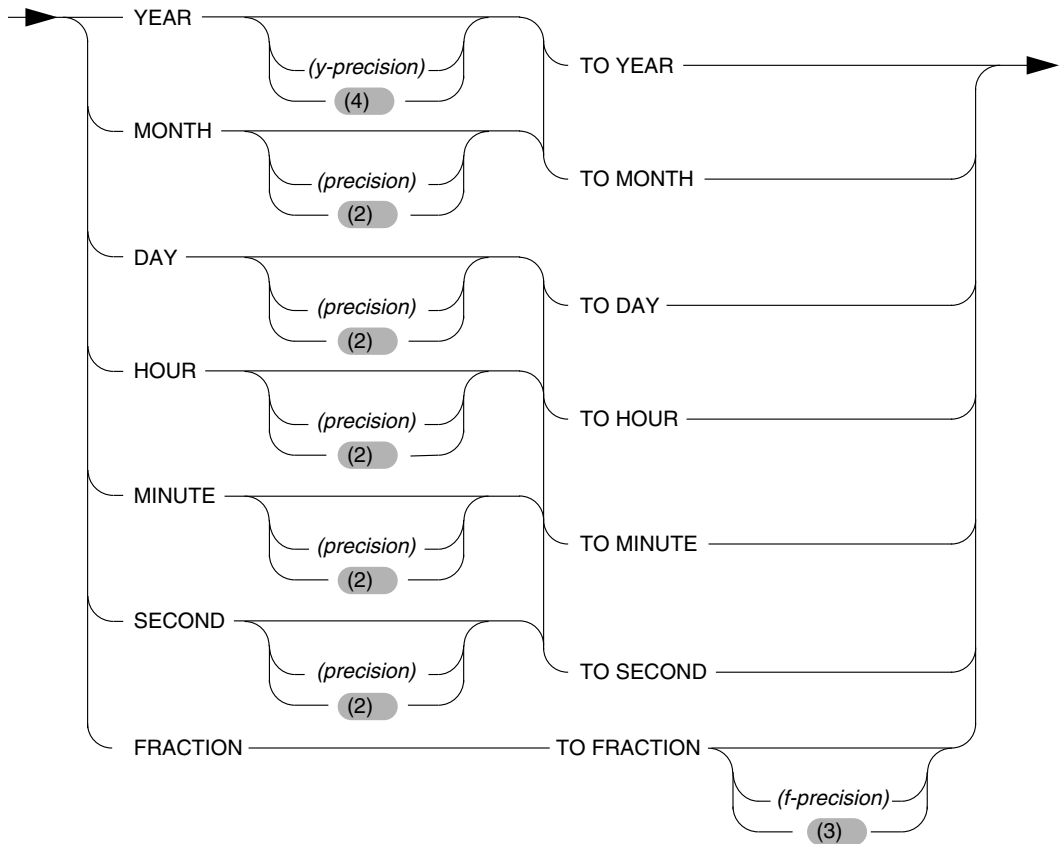
The *owner.name* is case sensitive. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 7-436. ♦

INTERVAL Field Qualifier

Purpose

Use the INTERVAL field qualifier to specify the units for an INTERVAL value. The INTERVAL field qualifier is used in the Data Type segment.

Syntax



<i>f-precision</i>	is the maximum number of digits you can use in the fraction field. The default is three; the maximum is five.
<i>precision</i>	is the number of digits in the largest number of months, days, hours, or minutes that the interval can hold. The default is two; the maximum is nine.
<i>y-precision</i>	is the number of digits in the largest number of years that the interval can hold. The default is four; the maximum is nine.

Usage

The following examples of an INTERVAL data type are both of type YEAR TO MONTH. The first example can hold an interval of up to 999 years and 11 months, since 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 intend for 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.

Figure 7-94
Examples of INTERVAL qualifiers

```
YEAR (5) TO MONTH
```

```
DAY (5) TO FRACTION (2)
```

```
DAY TO DAY
```

```
FRACTION TO FRACTION (4)
```

References

In this manual, for information about using INTERVAL data in arithmetic and relational operations, see [“Range of Operations Using DATE, DATETIME, and INTERVAL”](#) on page 3-25.

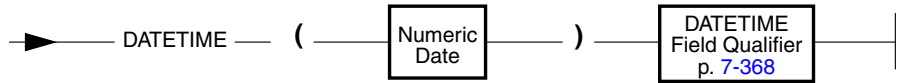
Literal DATETIME

Purpose

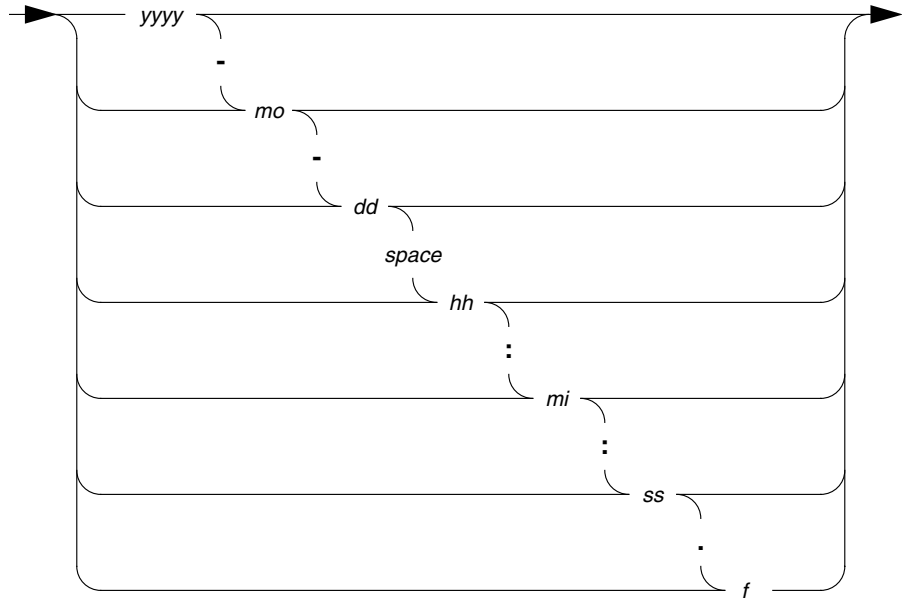
Use a literal DATETIME segment as a DATETIME value. The literal DATETIME segment is used in the following statements and segments:

- INSERT statement
- SELECT statement
- UPDATE statement
- Condition segment
- Expression segment

Syntax



Numeric Date



yyyy is the year in up to four digits. If you use two digits, 19 is assumed as the first part of the year, as in 1993.

mo is the month in two digits.

dd is the day in up to two digits.

space is, literally, a space made by pressing the spacebar.

hh is the hour in up to two digits.

<i>mi</i>	is the minute in up to two digits.
<i>ss</i>	is the second in up to two digits.
<i>f</i>	is the fraction of a second in up to five digits, depending on the precision given to the fractional portion in the INTERVAL qualifier.

Usage

Some examples of literal DATETIME values follow:

```
DATETIME (89-3-6) YEAR TO DAY
DATETIME (09:55:30.825) HOUR TO FRACTION
DATETIME (92-5) YEAR TO MONTH
```

Figure 7-95
Examples of literal DATETIME values

Here is an example of a literal DATETIME value used with the EXTEND function:

```
EXTEND (DATETIME (1989-8-1) YEAR TO DAY, YEAR TO MINUTE)
- INTERVAL (720) MINUTE (3) TO MINUTE
```

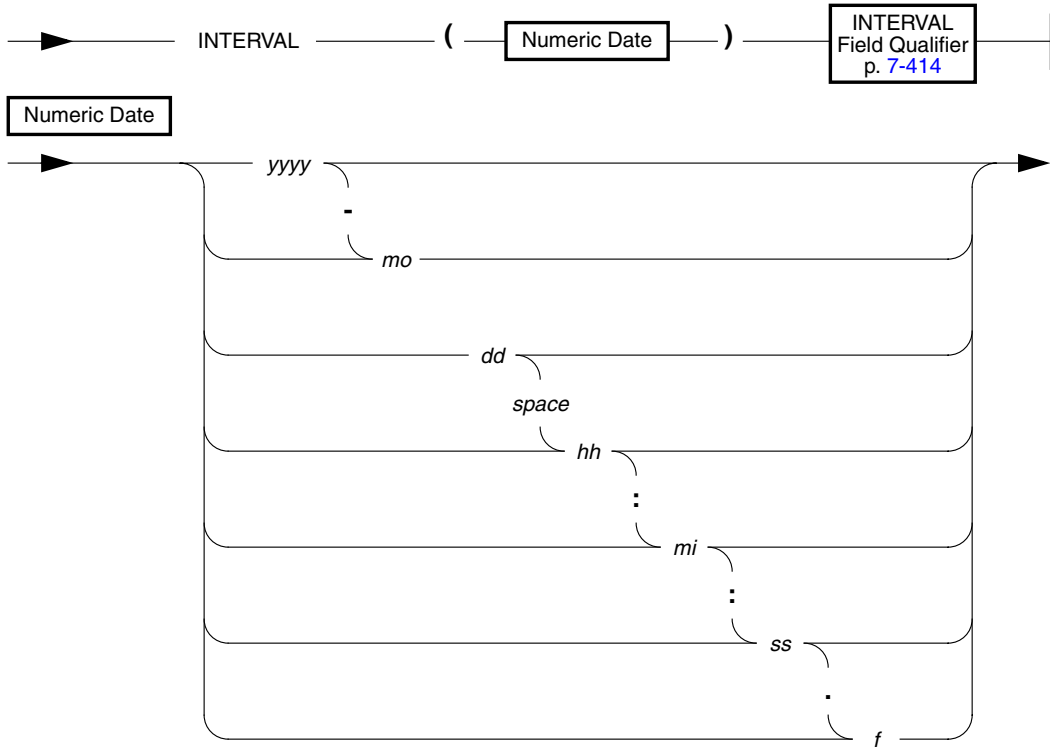
Literal INTERVAL

Purpose

The literal INTERVAL segment is used in the following statements and segments:

- INSERT statement
- UPDATE statement
- Condition segment
- Expression segment

Syntax



yyyy is the number of years. The maximum number of digits allowed is four, unless this is the first field and the precision is specified differently by the `INTERVAL` field qualifier.

mo is the number of months. The maximum number of digits allowed is two, unless this is the first field and the precision is specified differently by the `INTERVAL` field qualifier.

dd is the number of days. The maximum number of digits allowed is two, unless this is the first field and the precision is specified differently by the `INTERVAL` field qualifier.

space is, literally, a space made by pressing the spacebar.

<i>hh</i>	is the number of hours. The maximum number of digits allowed is two, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.
<i>mi</i>	is the number of minutes. The maximum number of digits allowed is two, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.
<i>ss</i>	is the number of seconds. The maximum number of digits allowed is two, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.
<i>f</i>	is the fraction of a second in up to five digits, depending on the precision given to the fractional portion in the INTERVAL field qualifier.

Usage

Some examples of literal INTERVAL values follow.

```
INTERVAL (3-6) YEAR TO MONTH  
INTERVAL (09:55:30.825) HOUR TO FRACTION  
INTERVAL (40 5) DAY TO HOUR
```

Figure 7-96
Examples of literal INTERVAL values

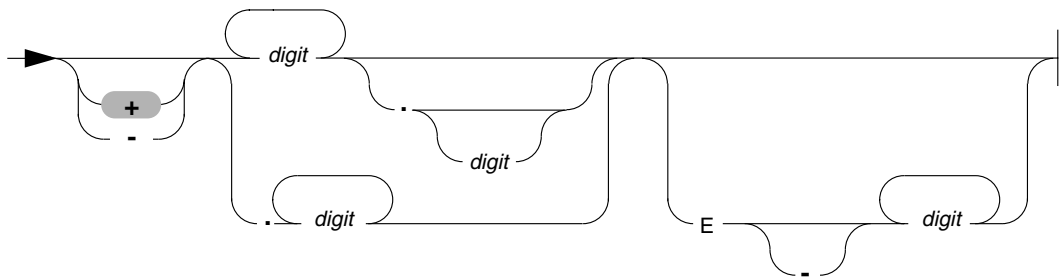
Literal Number

Purpose

A literal number is a constant and either an integer or noninteger (floating) number. The Literal Number segment is used in the following statements and segments:

- INSERT statement
- UPDATE statement
- Condition segment
- Expression segment

Syntax



digit is an integer from 0 to 9.

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 do not contain decimal points. Some examples of integers follow:

10 -27 25567

Floating and decimal numbers contain a decimal point and/or exponential notation. Some examples of floating and decimal numbers follow:

```
123.456  1.23456e2  123456.0e-3
```

When you use a literal number as a MONEY value, do not precede it with a money symbol or include commas.

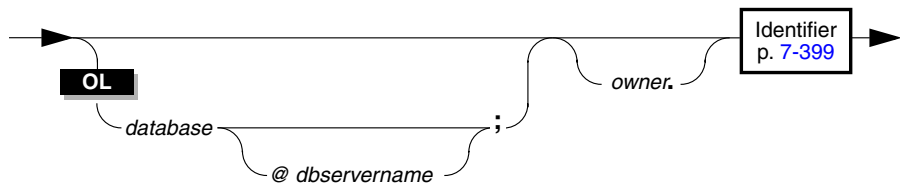
Procedure Name

Purpose

Use the Procedure Name segment wherever you see a reference to a procedure name in a syntax drawing. It appears in the following statements:

- CREATE PROCEDURE
- DROP PROCEDURE
- EXECUTE PROCEDURE

Syntax



database is the name of the database in which the procedure resides.

dbservername is the name of the IBM Informix OnLine database server that is home to *database*. The @ sign is a literal character that you must use to introduce the database server name.

owner is the user name of the owner of the procedure. If you are using an ANSI-compliant database, you must use the *owner.* convention for a procedure that you do not own.

Usage

The actual name of the procedure is an SQL identifier.

If you are creating the procedure, the *name* of the procedure must be unique within a database.

ANSI

If you are creating the procedure, the combination *owner.name* must be unique within a database.

The *owner.name* is case sensitive. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 7-436. ♦

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 in your procedure, any calls by that name are to the procedure. That is, you cannot use the system function within the statement block in which the procedure is defined.

As an example, the following procedure uses two **length** functions. The first time the procedure calls the **length** function, it is the SQL function named LENGTH. The second time the procedure calls the **length** function is within a BEGIN...END block in which **length** has been defined as a procedure. The second call to **length** actually uses the user-created procedure called **length**.

```
CREATE PROCEDURE test_len()
RETURNING INT, INT;

DEFINE c INT;
DEFINE d INT;
LET c = (SELECT length(fname) FROM customer
        WHERE customer_num = 101);

BEGIN
    DEFINE length PROCEDURE;
    LET d = length(5);
END

RETURN c, d;

END PROCEDURE;
```

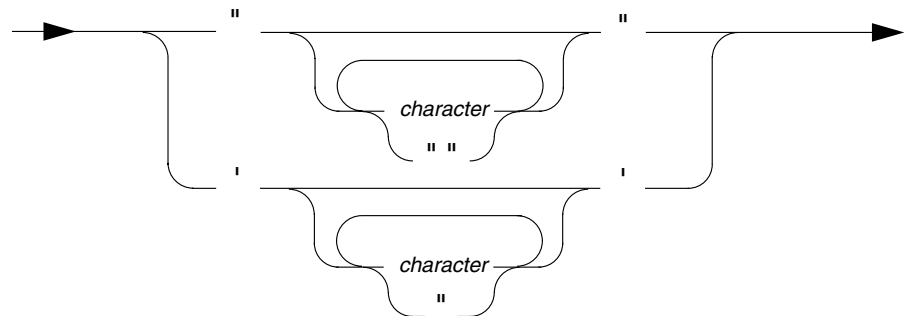
Quoted String

Purpose

The Quoted String segment is used in the following statements and segments:

- INSERT statement
- SELECT statement
- Condition segment
- Expression segment (in constant expressions)

Syntax



character is an ASCII character.

Usage

The constant must be written on a single line; that is, you cannot use embedded new lines.

Using Quotes in Strings

The single quote has no special significance in string constants delimited by double quotes. Likewise, the double quote has no special significance in strings delimited by single quotes. For example, the following strings are valid:

```
"Nancy's puppy jumped the fence"
'Billy told his kitten, "no!"'
```

You can include a double quote in a string by preceding the double quote with another double quote, as shown in the following string:

```
"Enter "y" to select this row"
```

DATETIME and INTERVAL Values as Strings

You can enter DATETIME and INTERVAL data in the literal forms described in the [“Literal DATETIME”](#) and [“Literal INTERVAL”](#) segments beginning on pages 7-416 and 7-419, 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 ("1992-5-4 10:12:11")
INSERT INTO manufact(lead_time) VALUES ("14")
```

The format of the value in the quoted string must match exactly the format specified by the qualifiers of the column. For the first case in the preceding example, `call_dtime` must be defined with the qualifiers YEAR TO MINUTE for the INSERT statement to be valid.

LIKE and MATCHES in a Condition

Quoted strings with the keyword LIKE or MATCHES in a condition can include wildcard characters. See the [“Condition”](#) segment beginning on page 7-345 for a complete description of how to use wildcard characters.

Inserting Values as Quoted Strings

If you are inserting a value that is a quoted string, you must follow these conventions:

- Enclose CHAR, VARCHAR, DATE, DATETIME, and INTERVAL values in quotation marks.
- Set DATE values in the *mm/dd/yyyy* format or in the format specified by DBSET, if set.
- For all statements except PREPARE, the maximum length for a quoted string is 256 bytes. The maximum length for a quoted string in a PREPARE statement is 2048 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.

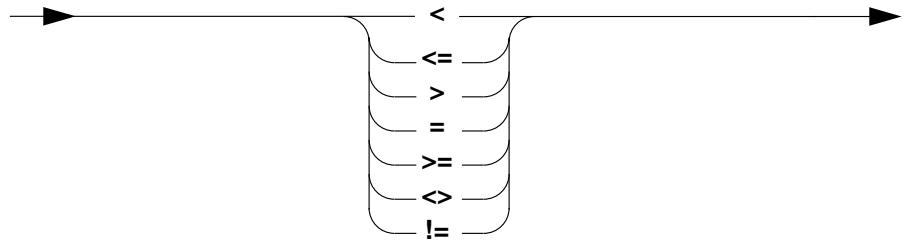
Relational Operator

Purpose

Use a relational operator to compare two expressions quantitatively. The Relational Operator segment is used in the Condition segment.

Syntax

A relational operator takes the following form:



<	means less than.
<=	means less than or equal to.
>	means greater than.
=	means equal to.
>=	means greater than or equal to.
<>	means not equal to.
!=	means 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 character expressions, *greater than* means *after* in ASCII collating order, where lowercase letters follow uppercase letters, and both follow numerals. The following chart contains the seven-bit ASCII collating order.

Num	Char	Num	Char	Num	Char
0	^@	43	+	86	V
1	^A	44	,	87	W
2	^B	45	-	88	X
3	^C	46	.	89	Y
4	^D	47	/	90	Z
5	^E	48	0	91	[
6	^F	49	1	92	\
7	^G	50	2	93]
8	^H	51	3	94	^
9	^I	52	4	95	_
10	^J	53	5	96	`
11	^K	54	6	97	a
12	^L	55	7	98	b
13	^M	56	8	99	c
14	^N	57	9	100	d
15	^O	58	:	101	e
16	^P	59	;	102	f
17	^Q	60	<	103	g
18	^R	61	=	104	h
19	^S	62	>	105	i
20	^T	63	?	106	j

(1 of 2)

Num	Char	Num	Char	Num	Char
21	^U	64	@	107	k
22	^V	65	A	108	l
23	^W	66	B	109	m
24	^X	67	C	110	n
25	^Y	68	D	111	o
26	^Z	69	E	112	p
27	esc	70	F	113	q
28	^\	71	G	114	r
29	^]	72	H	115	s
30	^^	73	I	116	t
31	^_	74	J	117	u
32		75	K	118	v
33	!	76	L	119	w
34	"	77	M	120	x
35	#	78	N	121	y
36	\$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	
39	'	82	R	125	}
40	(83	S	126	~
41)	84	T	127	del
42	*	85	U		

^X = CTRL-X

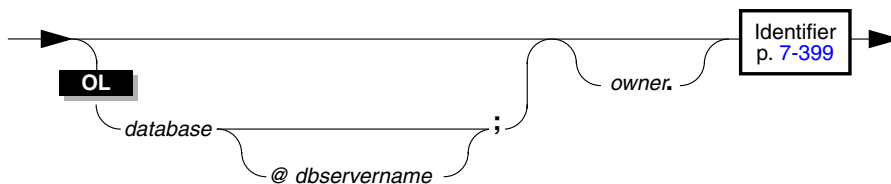
Synonym Name

Purpose

Use the Synonym Name segment wherever you see a reference to a synonym name in a syntax drawing. It appears in the following statements:

- CREATE AUDIT
- CREATE INDEX
- CREATE SYNONYM
- CREATE VIEW
- DELETE
- DROP AUDIT
- DROP SYNONYM
- DROP TABLE
- DROP VIEW
- INSERT
- LOCK TABLE
- RECOVER TABLE
- RENAME COLUMN
- REVOKE
- SELECT
- UNLOCK TABLE
- UPDATE
- UPDATE STATISTICS

Syntax



database is the name of the database in which the synonym resides.

dbservername is the name of the IBM Informix OnLine database server that is home to *database*. The @ sign is a literal character that you must use to introduce the database server name.

owner is the user name of the owner of the synonym. If you are using an ANSI-compliant database, you must use the *owner.* convention for a synonym that you do not own.

Usage

The actual name of the synonym is an SQL identifier.

If you are creating the synonym, the *name* of the synonym must be unique within a database.

If you are creating the synonym, the combination *owner.name* must be unique within a database.

The *owner.name* is case sensitive. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page [7-436](#). ♦

ANSI

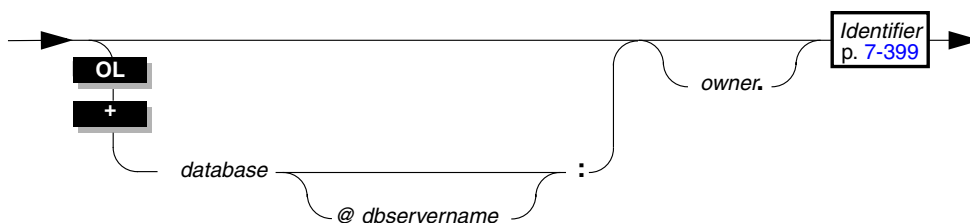
Table Name

Purpose

Use the Table Name segment in the following statements to specify the name of a table:

- ALTER TABLE
- CREATE AUDIT
- CREATE INDEX
- CREATE SYNONYM
- CREATE TABLE
- DELETE
- DROP AUDIT
- DROP TABLE
- GRANT
- INSERT
- LOCK TABLE
- RECOVER TABLE
- RENAME COLUMN
- REVOKE
- SELECT
- UNLOCK TABLE
- UPDATE
- UPDATE STATISTICS

Syntax



database is the name of the database in which the table resides.

dbservername is the name of the IBM Informix OnLine database server that is home to *database*. The @ sign is a literal character that you must use to introduce the database server name.

owner is the user name of the owner of the table. If you are using an ANSI-compliant database, you must use the *owner.* convention for tables that you do not own.

Usage

The following example shows a table specification:

```
empinfo@personnel:emp_names
```

If you are creating or renaming a table, you must make sure that the *name* of the table is unique within a database.

If you are creating or renaming a table, you must make sure that the combination of *owner* and *name* is unique within a database.

In an ANSI-compliant database, the table name must include *owner.* unless you are the owner. For system catalog tables, the owner is *informix.* ♦

ANSI

Case Sensitivity in ANSI-Compliant Databases

The database server shifts the owner name to uppercase letters before the statement is executed, unless the owner name is enclosed in quotes. Put quotes around the owner portion of a name if you want the owner to be read exactly as written. For example, the name **cathl** in the first statement that follows is upshifted to **CATHL** before it is used, while the name **nancy** in the second statement is not upshifted:

```
SELECT * FROM cathl.customer

SELECT * FROM "nancy".customer
```

There is no problem if you create a table with an implicit owner in uppercase letters and the owner's real login name is also in uppercase letters. For example, suppose that you are the user **BROWN** and you create a view with the following statement:

```
CREATE VIEW newcust AS
  SELECT fname, lname FROM customer WHERE state = "NJ"
```

You, **BROWN**, can run the following **SELECT** statements on the view:

```
SELECT * FROM brown.newcust

SELECT * FROM newcust

SELECT * FROM systables WHERE tablename = newcust
      AND owner = USER
```

In the first query in the preceding example, the database server automatically upshifts **brown** before the **SELECT** statement is executed. In the second query, the database server returns the owner name **BROWN** already upshifted. In the third query, **USER** returns the login name as it is stored—in this case, in uppercase letters.

If you are the user **nancy** and you use the following statement, the resulting view has the name **NANCY.njcust**:

```
CREATE VIEW nancy.njcust AS
  SELECT fname, lname FROM customer WHERE state = "NJ"
```

If you are **nancy** and you use the following statement, the resulting view has the name **nancy.njcust**:

```
CREATE VIEW "nancy".njcust AS
  SELECT fname, lname FROM customer WHERE state = "NJ"
```

The following SELECT statement fails because it tries to match the name **NANCY.njcust** to the actual owner and table name of **nancy.njcust**:

```
SELECT * FROM nancy.njcust
```



References

In the *IBM Informix Guide to SQL: Tutorial*, see the discussion of owner naming.

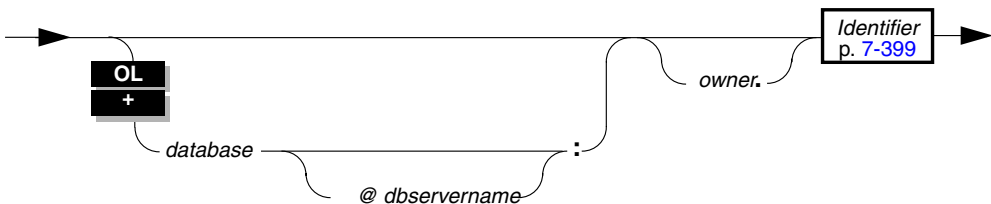
View Name

Purpose

Use the View Name segment in the following statements to specify the name of a view:

- CREATE SYNONYM
- CREATE VIEW
- DELETE
- DROP VIEW
- GRANT
- INSERT
- REVOKE
- SELECT
- UPDATE

Syntax



database is the name of the database in which the view resides.

dbservername is the name of the IBM Informix OnLine database server that is home to *database*. The @ sign is a literal character that you must use to introduce the database server name.

owner is the user name of the owner of the view. If you are using an ANSI-compliant database, you must use the *owner.* convention for views that you do not own.

ANSI

Usage

The use of the prefix *owner.* is optional; however, if you use it, the database server does check *owner* for accuracy. If you are creating a view, the *name* of the view must be unique among all the tables, synonyms, and views that already exist in the database.

If you are creating a view, the *owner.view-name* must be unique among all the tables, synonyms, and views that already exist in the database.

The *owner.name* is case sensitive. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page [7-436](#). ♦

References

In the *IBM Informix Guide to SQL: Tutorial*, see the discussions of views and security.

Stored Procedures and SPL

In This Chapter	8-5
Introduction to Stored Procedures and SPL	8-5
What You Can Do with Stored Procedures	8-6
Relationship Between SQL and a Stored Procedure	8-6
Creating and Using Stored Procedures	8-7
Creating a Procedure Using DB-Access	8-7
Creating a Procedure Using an Embedded-Language Product	8-8
Commenting and Documenting a Procedure	8-8
Diagnosing Compile-Time Errors	8-9
Finding Syntax Errors in a Procedure Using DB-Access	8-9
Finding Syntax Errors in a Procedure Using an Embedded-Language Product	8-9
Looking at Compile-Time Warnings	8-10
Generating the Text or Documentation.	8-11
Looking at the Procedure Text	8-11
Looking at the Procedure Documentation	8-11
Executing a Procedure	8-12
Debugging a Procedure	8-14
Re-creating a Procedure	8-16
Privileges on Stored Procedures	8-16
Privileges at Creation.	8-16
Privileges at Execution	8-17
Privileges and Owner-Privileged Procedures	8-17
Privileges and DBA-Privileged Procedures	8-18
Privileges and Nested Procedures	8-18
Revoking Privileges	8-19

Variables and Expressions	8-19
Variables	8-19
Format of Variables	8-19
Global and Local Variables	8-20
Defining Variables	8-20
Data Types for Variables	8-20
Scope of Variables.	8-21
Variable/Keyword Ambiguity	8-22
Expressions	8-23
Assigning Values to Variables	8-25
Program Flow Control	8-26
Branching	8-26
Looping	8-27
Function Handling.	8-28
Calling Procedures Within a Procedure	8-28
Running an Operating System Command from Within a Procedure	8-28
Recursively Calling a Procedure.	8-29
Passing Information into and out of a Procedure.	8-29
Returning Results	8-29
Specifying Return Values	8-29
Returning the Value	8-30
Returning More Than One Set of Values from a Procedure.	8-30
Exception Handling.	8-32
Trapping an Error and Recovering	8-32
Scope of Control of an ON EXCEPTION Statement	8-33
User-Generated Exceptions.	8-34
Simulating SQL Errors	8-34
Using RAISE EXCEPTION to Exit Nested Code	8-35
SPL Statement Syntax	8-36
CALL	8-37
CONTINUE	8-40
DEFINE	8-42
EXIT	8-50
FOR	8-52
FOREACH	8-56
IF.	8-60

LET	8-64
ON EXCEPTION	8-67
RAISE EXCEPTION	8-73
RETURN	8-75
SYSTEM	8-78
TRACE	8-80
WHILE	8-84

In This Chapter

Stored procedures are effective tools that you can use to control SQL activity. This chapter provides instruction on how to write stored procedures using SQL and the Informix Stored Procedure Language (SPL). To help you learn how to write them, examples of working stored procedures are provided.

The syntax for each of the SPL statements is provided at the end of the chapter. Accompanying the syntax for each statement are usage notes and examples pertinent to that statement.

Introduction to Stored Procedures and SPL

To SQL, a stored procedure is a user-defined function. Anyone who has Resource privilege on a database can create a stored procedure. Once the stored procedure is created, it is stored in an executable format in the database as an object of the database. You can use stored procedures to perform any function you can perform in SQL, as well as expand what you can accomplish with SQL alone.

You write a stored procedure using SQL and SPL statements. SPL statements only can be used inside `CREATE PROCEDURE` and `CREATE PROCEDURE FROM` statements. These statements are available with DB-Access and the embedded-language products.

What You Can Do with Stored Procedures

You can accomplish a wide range of objectives with stored procedures, including improving database performance, simplifying the writing of applications, and limiting or monitoring access to data.

Since a stored procedure is stored in an executable format, you can use it to execute frequently repeated tasks to improve performance. Executing a stored procedure rather than straight SQL code allows you to bypass repeated parsing, validity checking, and query optimization.

Since a stored procedure is an object in the database, it is available to every application running on the database. Several applications can use the same stored procedure, so development time for applications is reduced.

Since you can write a stored procedure to be run with DBA privilege by a user who does not have DBA privilege, you can limit and control access to data in the database. Alternatively, a stored procedure can monitor what users access certain tables or data.

Relationship Between SQL and a Stored Procedure

You can use stored procedures to supply values to data manipulation statements of SQL. For example, these values can be inserted into a table or used as part of a condition in a query or an UPDATE statement, and so on.

Alternatively, you can use stored procedures to hide SQL statements. Implementing stored procedures can simplify tasks for a database user. Rather than having all users learn how to use SQL, one experienced SQL user can write a stored procedure to encapsulate an SQL activity and let others know that the procedure is stored in the database and that they can execute it.

Creating and Using Stored Procedures

To write a stored procedure, put the SQL statements that you want to be run as part of the procedure inside the statement block in a CREATE PROCEDURE statement. Use the SPL statements to control the flow of operation within the procedure. These additional statements include IF, FOR, and others, and are described at the end of this chapter. The CREATE PROCEDURE and CREATE PROCEDURE FROM statements are described in [Chapter 7, "Syntax."](#)

Creating a Procedure Using DB-Access

To create a stored procedure using DB-Access, issue the CREATE PROCEDURE statement, including all of the statements that are part of the procedure in the statement block. For example, to create a procedure that reads a customer address, you can use a statement such as the one in [Figure 8-1](#).

Figure 8-1
Procedure that reads from the customer table

```
CREATE PROCEDURE read_address (lastname CHAR(15)) -- one argument
  RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),CHAR(2), CHAR(5); -- 6
items

  DEFINE p_lname,p_fname, p_city CHAR(15); --define each procedure
variable
  DEFINE p_add CHAR(20);
  DEFINE p_state CHAR(2);
  DEFINE p_zip CHAR(5);

  SELECT fname, address1, city, state, zipcode
     INTO p_fname, p_add, p_city, p_state, p_zip
    FROM customer
   WHERE lname = lastname;

  RETURN p_fname, lastname, p_add, p_city, p_state, p_zip; --6 items
END PROCEDURE

DOCUMENT "This procedure takes the last name of a customer as", --brief
description
  "its only argument. It returns the full name and address of",
  "the customer."
WITH LISTING IN "/acctng/test/listfile" -- compile-time warnings go here
; -- end of the procedure read_address
```

Creating a Procedure Using an Embedded-Language Product

To create a stored procedure using an embedded-language product, put the text of the CREATE PROCEDURE statement in a file. Use the CREATE PROCEDURE FROM statement and refer to that file to compile the procedure. For example, to create a procedure to read a customer name, you can use a statement such as the one in [Figure 8-1](#) and store it in a file. If the file is named **read_add_source**, the following statement compiles the **read_address** procedure:

```
CREATE PROCEDURE FROM "read_add_source";
```

[Figure 8-2](#) shows how the previous SQL statement looks in an ESQL/C program.

Figure 8-2

SQL statement that compiles and stores the read_address procedure in an ESQL/C program

```
/* This program creates whatever procedure is in *  
 * the file "read_add_source".  
 */  
#include <stdio.h>  
$include sqlca;  
$include sqllda;  
$include datetime;  
/* Program to create a procedure from the pwd */  
  
main()  
{  
$ database play;  
$create procedure from "read_add_source";  
}
```

Commenting and Documenting a Procedure

The **read_address** procedure in [Figure 8-1](#) includes comments and a DOCUMENT clause. The comments are incorporated into the text of the procedure. Any characters following a double hyphen (--) are considered to be a comment. The double hyphen can be used at the beginning or middle of a line.

To include a separate summary of the procedure, use the DOCUMENT clause. You can extract the data in the DOCUMENT clause by querying the **sysprocbody** system catalog table. See the section [“Looking at the Procedure Documentation”](#) later in this chapter for more information about reading the DOCUMENT clause.

Diagnosing Compile-Time Errors

When you issue a CREATE PROCEDURE or CREATE PROCEDURE FROM statement, the statement fails if a syntax error is in the body of the procedure. The database server stops processing the text of the procedure and returns the location of the error.

Finding Syntax Errors in a Procedure Using DB-Access

If a procedure created using DB-Access has a syntax error, when you choose the Modify option of the QUERY-LANGUAGE Menu, the cursor is situated on the offending syntax.

Finding Syntax Errors in a Procedure Using an Embedded-Language Product

If a procedure created using an embedded-language product has a syntax error, the CREATE PROCEDURE statement fails. The database server sets the SQLCODE field of the SQLCA to a negative number and sets the fifth element of the SQLERRD array to the character offset into the file. The particular fields of the SQLCA for each product are shown in the following table.

ESQL/C	ESQL/COBOL
sqlca.sqlcode SQLCODE	SQLCODE OF SQLCA
sqlca.sqlerrd[4]	SQLERRD[5] OF SQLCA

Figure 8-3 shows how to trap for a syntax error when you are creating a procedure. It also shows how to display the offset into the file where the error occurred.

Figure 8-3
Checking for failure when creating a procedure using ESQL/C

```
#include <stdio.h>
#include sqlca;
#include sqllda;
#include datetime;
/* Program to create a procedure from procfile in pwd */

main()
{
long char_num;

$ database play;
$create procedure from "procfile";
if (sqlca.sqlcode != 0 )
{
printf("\nSqlca.sqlcode = %ld\n", sqlca.sqlcode);
char_num = sqlca.sqlerrd[4];
printf("\nError in creating read_address. Check character position
%ld\n",
char_num);
```

In [Figure 8-3](#), if the CREATE PROCEDURE FROM statement fails, the program displays a message in addition to the character position at which the syntax error was detected.

Looking at Compile-Time Warnings

If the database server detects a potential problem but the procedure is syntactically correct, a warning is placed in the listing file. You can use this listing to check for potential problems with a procedure before you execute it.

To obtain the listing of compile-time warnings for your procedure, use the WITH LISTING IN clause in your CREATE PROCEDURE statement, as shown in the following example.

```
CREATE PROCEDURE read_address (lastname CHAR(15)) -- one argument
RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15), CHAR(2), CHAR(5); -- 6
items
.
.
.
WITH LISTING IN "/acctng/test/listfile" -- compile-time warnings go here
; -- end of the procedure read_address
```


If you are using IBM Informix STAR or IBM Informix NET, the listing file is created on the machine on which the database resides. If you provide an absolute pathname and filename for the file, the file is created where you specify. If you provide a relative pathname for the listing file, the file is created in your home directory on that machine. (If you do not have a home directory, the file is created in the root directory.)

After you create the procedure, you can view the file specified in the WITH LISTING IN clause to see the warnings it contains.

Generating the Text or Documentation

Once you create the procedure, it is stored in the **sysprocbody** system catalog table. The **sysprocbody** system catalog table contains the executable procedure, as well as the text of the original CREATE PROCEDURE statement and the documentation text.

Looking at the Procedure Text

To generate the text of the procedure, select the data column from the **sysprocbody** system catalog table. The following SELECT statement reads the **read_address** procedure text.

```
SELECT data FROM informix.sysprocbody
  WHERE datakey = "T" -- find text lines
  AND
  procid = (SELECT procid FROM informix.sysprocedure
            WHERE informix.sysprocedure.procname = "read_address")
```

Looking at the Procedure Documentation

If you want to view only the documenting text of the procedure, you can use the following SELECT statement to read the documentation string. The documentation lines are those in the DOCUMENT clause of the CREATE PROCEDURE statement.

```
SELECT data FROM informix.sysprocbody
  WHERE datakey = "D" -- find documentation lines
  AND
  procid = (SELECT procid FROM informix.sysprocedures
            WHERE informix.sysprocedure.procname = "read_address")
```

Executing a Procedure

There are three ways to execute a procedure. You can use the SQL statement EXECUTE PROCEDURE or either the LET or CALL SPL statement.

You can execute a procedure with the EXECUTE PROCEDURE statement. To run the **read_address** procedure to see the full name and address of a customer named "Putnum," use the following statement:

```
EXECUTE PROCEDURE read_address ("Putnum");
```

Since the **read_address** procedure returns values, if you are executing a procedure from an embedded-language program or from another procedure, you must use an INTO clause with host variables to receive the data. For example, executing the **read_address** procedure in an ESQL/C program is accomplished with the code segment shown in [Figure 8-4](#).

Figure 8-4
Executing a procedure using ESQL/C

```
#include <stdio.h>
#include sqlca;
#include sqllda;
#include datetime;
/* Program to execute a procedure in the database named "play" */

main()
{
  $ char lname[16], fname[16], address[21];
  $ char city[16], state[3], zip[6];

  $ database play;
  $EXECUTE PROCEDURE read_address ("Putnum")
    INTO $lname, $fname, $address, $city, $state, $zip;
  if (sqlca.sqlcode != 0 )
    printf("\nFailure on execute");
}
```

If you are executing a procedure within another procedure, you can use the CALL or LET statement to run the procedure. To use the CALL statement with the **read_address** procedure, you can use the code shown in [Figure 8-5](#).

Figure 8-5*Calling a procedure inside another procedure with the CALL statement*

```

CREATE PROCEDURE address_list ()

    DEFINE p_lname, p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    .
    .
    .
    CALL read_address ("Putnum") RETURNING p_fname, p_lname,
        p_add, p_city, p_state, p_zip;
    .
    .
    .
    -- use the returned data some way
END PROCEDURE;

```

Figure 8-6 provides an example of using the LET statement to assign values to procedural variables through a procedure call.

Figure 8-6*Assigning values from a procedure call with a LET statement*

```

CREATE PROCEDURE address_list ()

    DEFINE p_lname, p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    .
    .
    .
    LET p_fname, p_lname, p_add, p_city, p_state, p_zip = read_address
("Putnum");
    .
    .
    .
    -- use the returned data some way
END PROCEDURE;

```

Debugging a Procedure

Once you successfully create and run a procedure, you can encounter logical errors. If logical errors are in the procedure, use the TRACE statement to help you find the errors. You can trace the values of the following procedural entities:

- Variables
- Procedure arguments
- Return values
- SQL error codes
- ISAM error codes

To generate a listing of traced values, first use the SQL statement SET DEBUG FILE to name the file that is to contain the traced output. When you create your procedure, include the TRACE statement in one of its forms.

There are three ways to specify the form of TRACE output:

TRACE ON traces all statements except SQL statements. The contents of variables are printed before being used. Procedure calls and returned values are traced as well.

TRACE traces only the procedure calls and returned values.
PROCEDURE

TRACE *expres-* prints a literal or an expression. If necessary, the value of the
sion expression is calculated before being sent to the file.

Figure 8-7 shows how you can use the TRACE statement with a version of the **read_address** procedure. Several of the SPL statements shown in this example have not been discussed, but the entire example demonstrates how the TRACE statement can help you monitor execution of the procedure.

Figure 8-7*Using the TRACE statement to monitor execution of a procedure*

```

CREATE PROCEDURE read_many (lastname CHAR(15))
RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),CHAR(2), CHAR(5);

    DEFINE p_lname,p_fname, p_city CHAR(15);
    DEFINE p_add CHAR(20);
    DEFINE p_state CHAR(2);
    DEFINE p_zip CHAR(5);
    DEFINE lcount, i INT;

    LET lcount = 1;

    TRACE ON; -- Every expression will be traced from here on
    TRACE "Foreach starts"; -- A trace statement with a
literal
    FOREACH
    SELECT fname, lname, address1, city, state, zipcode
        INTO p_fname, p_lname, p_add, p_city, p_state, p_zip
        FROM customer
        WHERE lname = lastname
    RETURN p_fname, p_lname, p_add, p_city, p_state, p_zip WITH RESUME;
    LET lcount = lcount + 1; -- count of returned addresses
    END FOREACH;

    TRACE "Loop starts"; -- Another literal
    FOR i IN (1 TO 5)
        BEGIN
            RETURN i , i+1, i*i, i/i, i-1,i with resume;
        END
    END FOR;

END PROCEDURE

;

```

Each time you execute the traced procedure, entries are added to the file you specified using the SET DEBUG FILE statement. To see the debug entries, view the output file using any text editor.

The following list contains some of the output generated by the procedure in [Figure 8-7](#). Next to each traced statement is an explanation of its contents.

TRACE ON	echoed TRACE ON statement.
TRACE Foreach starts	traced expression, in this case, the literal string Foreach starts.
start select cursor	notification that a cursor is opened to handle a FOREACH loop.
select cursor iteration	notification of the start of each iteration of the select cursor.

`expression: (+lcount, 1)` the encountered expression, (lcount+1) evaluates to 2.

`let lcount = 2` each LET statement is echoed with the value.

Re-creating a Procedure

If a procedure exists in a database, you must drop it explicitly using the DROP PROCEDURE statement before you can create another procedure with the same name. If you debug your procedure and attempt to use the CREATE PROCEDURE statement with the same procedure name again, the attempt fails unless you first drop the existing procedure from the database.

Privileges on Stored Procedures

There are two types of procedures: DBA-privileged and owner-privileged. The creator of the procedure determines the type of procedure it is; the procedure type affects the privileges of the user executing the procedure.

A procedure resides in the database in which it is created. Since it is a database object, a user requires certain privileges to create and execute a procedure. With DBA-privileged procedures, you can use the GRANT SQL statement to give another user EXECUTE privilege on the procedure.

With owner-privileged procedures, you also must use the GRANT statement to give users privileges on objects referenced by the procedure (if those objects are not owned by the owner of the procedure and if the procedure owner does not have privileges with the WITH GRANT OPTION statement).

Privileges at Creation

As with a table or a view, you must have Resource privilege on the database to create a procedure. The existence of other database objects (tables, views, and so on) is not checked when the procedure is created.

By default, when an owner-privileged procedure is created, PUBLIC has Execute privilege. When a DBA-procedure is created, only the owner and other users that have DBA status have Execute privilege by default.

In an ANSI-compliant database, only the owner and other users that have DBA status have Execute privilege by default on both DBA- and owner-privileged procedures.

Privileges at Execution

The owner of a procedure, a user with DBA status, or a user with Execute privilege can execute the procedure. The database server checks that the user executing the procedure has the correct privileges on the underlying objects at execution.

Any user who executes a procedure is automatically granted Resource privilege on the database for the duration of the procedure.

Privileges and Owner-Privileged Procedures

When an owner-privileged procedure is executed, the existence of referenced objects is checked. In addition, privileges on referenced objects are checked when the procedure is executed.

Any user executing a procedure must have the appropriate privileges granted to them by the owner of the procedure. If you execute a procedure that references only objects that you own, there cannot be any privilege conflicts. If you do not own the referenced objects and you execute a procedure that contains SELECT statements and CREATE TABLE statements, you must have Select privilege to run the procedure without generating errors. Alternatively, the owner of the procedure must have the appropriate privileges with the WITH GRANT OPTION statement. In [Figure 8-7](#), the owner of the procedure must have the Select privilege with the WITH GRANT option. When you use the GRANT statement to give another user Execute privilege on a procedure that you own, you implicitly grant all appropriate privileges on objects that you own and that are referenced in the procedure.

Unqualified objects created in the course of the procedure are owned by the owner of the procedure, not the user running the procedure. For example, the following lines in an owner-privileged stored procedure create two tables. If this procedure is owned by **tony** and a user **marty** runs the procedure, the first table, **gargantuan**, is owned by **tony**. The second table, **tiny**, is owned by **libby**.

```
CREATE PROCEDURE tryit ()
.
.
.
CREATE TABLE gargantuan (col1 INT, col2 INT, col3 INT);
CREATE TABLE libby.tiny (col1 INT, col2 INT, col3 INT);

END PROCEDURE
```

Privileges and DBA-Privileged Procedures

When a DBA-privileged procedure is executed, the user executing the procedure assumes the privileges of a DBA for the duration of the procedure. A DBA-privileged procedure acts as if the current user is first granted DBA privilege, then executes each statement of the procedure manually, and finally has DBA privilege revoked.

Objects created in the course of running a DBA procedure are owned by the user running the procedure, unless the data definition statement in the procedure explicitly names the owner to be someone else.

Only a DBA or a user with DBA privilege can create a DBA-privileged procedure.

Privileges and Nested Procedures

DBA-privileged status is not inherited by a called procedure. If a DBA-privileged procedure executes an owner-privileged procedure, the owner-privileged procedure does not run as a DBA procedure. If an owner-privileged procedure calls a DBA-privileged procedure, the statements within the DBA-privileged procedure execute as they would within any DBA-privileged procedure.

Revoking Privileges

The owner of a procedure can revoke Execute privilege from a user. If a user loses Execute privilege on a procedure, the Execute privilege also is revoked from all users who were granted Execute privilege by that user.

Variables and Expressions

This section discusses how to define and use variables in SPL. The differences between SPL and SQL expressions also is covered here.

Variables

You can use variables in a stored procedure in several ways. A variable can be used in a database query or other SQL statement wherever a constant is expected. You can use a variable with SPL statements to assign and calculate values, keep track of the number of rows returned from a query, and execute a loop, as well as handle other tasks.

The value of a variable is held in memory; the variable is not a database object. Hence, rolling back a transaction does not restore values of procedural variables.

Format of Variables

A variable follows the rules of an SQL identifier. (See page [7-399](#) for the syntax and notes for an identifier.) Once a variable is defined, you can use it as appropriate anywhere in the procedure.

If you are using an embedded-language product, you do not have to set off the variable with a special symbol (unlike host variables in an embedded-language product).

Global and Local Variables

You can define a variable to be either local or global. A variable is local by default. The differences between the two types are outlined as follows:

- | | |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Local | A local variable is available only within the procedure in which it is defined. Local variables do not allow a default value to be assigned at compile time. |
| Global | A global variable is available to other procedures run by the same user session in the same database. The values of global variables are stored in memory. The global environment is the memory used by all the procedures run within a given session on a given database server, such as all procedures run by an embedded-language program or in a DB-Access session. The values of the variables are lost when the session ends. |

Global variables require a default value to be assigned at compile time.

The first definition of a global variable puts the variable into the global environment. Subsequent definitions of the same variable, in different procedures, simply bind the variable to the global environment.

Defining Variables

You define variables using the DEFINE statement. If you list a variable in the argument list of a procedure, the variable is defined implicitly and you do not need to define it formally with the DEFINE statement. You must assign a value, which may be null, to a variable before you can use it.

Data Types for Variables

You can define a variable as any of the data types available for columns in a database table except SERIAL. The following example shows several cases of defined procedural variables:

```
DEFINE x INT;  
DEFINE name CHAR(15);  
DEFINE this_day DATETIME YEAR TO DAY ;
```

If you define a variable for TEXT or BYTE data, the variable does not actually contain the data; instead, it serves as a pointer to the data. However, you use this procedural variable as you would use any other procedural variable. When you define a TEXT or BYTE variable, you must use the word REFERENCES, which emphasizes that these variables do not contain the data, but simply reference the data. The following example shows the definition of a TEXT and a BYTE variable:

```
DEFINE ttt REFERENCES TEXT;
DEFINE bbb REFERENCES BYTE;
```

Scope of Variables

A variable is valid within the statement block in which it is defined. It is valid within statement blocks nested within that statement block as well, unless it is masked by a redefinition of a variable with the same name.

In the beginning of the procedure in [Figure 8-8](#), the integer variables *x*, *y*, and *z* are defined and initialized. The BEGIN and END statements mark a nested statement block in which the integer variables *x* and *q* are defined, as well as the CHAR variable *z*. Within the nested block, the redefined variable *x* masks the original variable *x*. After the END statement, which marks the end of the nested block, the original value of *x* is again accessible.

Figure 8-8

Masking of variables with nested statement blocks

```
CREATE PROCEDURE scope()
  DEFINE x,y,z INT;
  LET x = 5; LET y = 10;
  LET z = x + y; --z is 15
  BEGIN
    DEFINE x, q INT; DEFINE z CHAR(5);
    LET x = 100;
    LET q = x + y; -- q = 110
    LET z = "silly"; -- z receives a character value
  END
  LET y = x; -- y is now 5
  LET x = z; -- z is now 15, not "silly"
END PROCEDURE
```

Variable/Keyword Ambiguity

If you define a variable as a keyword, ambiguities can occur. The following rules for identifiers help you avoid ambiguities for variables, procedure names, and system function names.

- Defined variables take the highest precedence.
- Procedures defined as such in a DEFINE statement take precedence over SQL functions.
- SQL functions take precedence over procedures that exist but are *not* identified as procedures (using the DEFINE statement).

In some cases, you must change the name of your variable. For example, you cannot define a variable with the name **count** or **max**, because these are the names of aggregate functions. The keywords that can be used ambiguously are listed in [“Identifier” on page 7-399](#).

Variables and Column Names

If you use the same identifier for a procedural variable as for a column name, the database server assumes that an instance of the identifier is a variable. Qualify the column name with the table name to use the identifier as a column name. In the following example, the procedure variable **lname** is the same as the column name. In the following SELECT statement, **customer.lname** is a column name and **lname** is a variable name:

```
CREATE PROCEDURE table_test()  
  
    DEFINE lname CHAR(15);  
    LET lname = "Miller";  
  
    SELECT customer.lname FROM customer INTO lname  
        WHERE customer_num = 502.  
    .  
    .  
    .
```

Variables and SQL Functions

If you use the same identifier for a procedural variable as for an SQL function, the database server assumes that an instance of the identifier is a variable and disallows the use of the SQL function. You cannot use the SQL function within the block of code in which the variable is defined. The following example shows a block within a procedure in which a variable called **user** is defined. This definition disallows the use of the USER function while in the BEGIN...END block.

```
CREATE PROCEDURE user_test()
  DEFINE name CHAR(10);
  DEFINE name2 CHAR(10);
  LET name = user; -- the SQL function

  BEGIN
  DEFINE user CHAR(15); -- disables user function
  LET user = "Miller";
  LET name = user; -- assigns "Miller" to variable name

  END
  .
  .
  .
  LET name2 = user; -- SQL function again
```

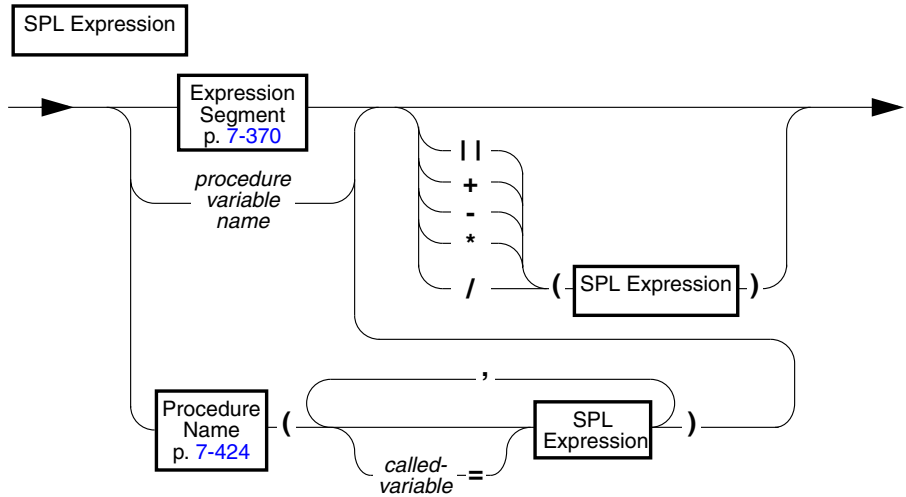
Procedure Names and SQL Functions

For information about ambiguities between procedure names and SQL function names, see [“Procedure Name” on page 7-424](#).

Expressions

You can use any SQL expression in a stored procedure except for an aggregate expression. The complete syntax and notes for SQL expressions are described on page [7-370](#).

In addition to being able to use SQL expressions, you can use procedure variables and values returned by procedures as expressions or as parts of expressions.



called variable is the name of one of the arguments expected by the called procedure.

procedure variable name is the name of a valid variable defined in the procedure

Some examples of typical SPL expressions are listed in the following example:

```
var1
var1 + var2 + 5
read_address("Miller")
read_address(lastname = "Miller")
get_duedate(acct_num) + 10 UNITS DAY
fname || " " || lname
"(415) " || get_phonenum(cust_name)
```

Assigning Values to Variables

There are four ways to assign a value to a procedure variable:

- Using a LET statement
- Using a SELECT...INTO statement
- Using a CALL statement with a procedure that has a RETURNING clause
- Using an EXECUTE PROCEDURE...INTO statement

Use the LET statement to assign a value to one or more variables. The following example illustrates several forms of the LET statement:

```
LET a = b + a;
LET a, b = c, d;
LET a, b = (SELECT fname, lname FROM customer
           WHERE customer_num = 101);
LET a, b = read_name(101);
```

Use the SELECT statement to assign a value directly from the database to a variable. The statement in the following example accomplishes the same task as the third LET statement in the previous example:

```
SELECT fname, lname into a, b FROM customer
       WHERE customer_num = 101
```

Use the CALL or EXECUTE PROCEDURE statements to assign values returned by a procedure to one or more procedural variables. Both statements in the following example return the full address from the procedure `read_address` into the specified procedural variables:

```
EXECUTE PROCEDURE read_address("Smith")
  INTO p_fname, p_lname, p_add, p_city, p_state, p_zip;

CALL read_address("Smith")
  RETURNING p_fname, p_lname, p_add, p_city, p_state, p_zip;
```

Program Flow Control

SPL contains several statements that enable you to control the flow of your stored procedure and to make decisions based on data obtained at run time. These program-flow-control statements are described briefly in this section of the chapter. Their syntax and complete descriptions are provided in [“SPL Statement Syntax” on page 8-36](#).

Branching

Use an IF statement to form a logic branch in a stored procedure. An IF statement first evaluates a condition and, if the condition is true, the statement block contained in the THEN portion of the statement is executed. If the condition is not true, execution falls through to the next statement, unless an ELSE clause or ELIF (else if) clause is included in the IF statement.

[Figure 8-9](#) shows an example of an IF statement.

Figure 8-9
Using an IF statement

```
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
```


Looping

There are three methods of looping in SPL, accomplished with one of the following statements:

- FOR initiates a controlled loop. Termination is guaranteed.
- FOREACH allows you to select and manipulate more than one row from the database. It declares and opens a cursor implicitly.
- WHILE initiates a loop. Termination is not guaranteed.

There are four ways to leave a loop, accomplished with one of the following statements:

- CONTINUE skips the remaining statements in the present, identified loop and starts the next iteration of that loop.
- EXIT exits the present, identified loop. Execution resumes at the first statement after the loop.
- RETURN exits the procedure. If a return value is specified, that value is returned upon exit.
- RAISE EXCEP- exits the loop if the exception is not trapped (caught) in the
TION body of the loop.

See [“SPL Statement Syntax” on page 8-36](#) for more information concerning the syntax and use of these statements.

Function Handling

You can call procedures, as well as run operating system commands, from within a procedure.

Calling Procedures Within a Procedure

Use a CALL statement or the SQL statement EXECUTE PROCEDURE to execute a procedure from a procedure. [Figure 8-10](#) shows a call to the `read_name` procedure using a CALL statement.

Figure 8-10
Calling a procedure with a CALL statement

```
CREATE PROCEDURE call_test()
  RETURNING CHAR(15), CHAR(15);

  DEFINE fname, lname CHAR(15);
  CALL read_name("Putnum") RETURNING fname, lname;

  IF fname = "Eileen" THEN RETURN "Jessica", lname;
  ELSE RETURN fname, lname;
  END IF
END PROCEDURE
```

Running an Operating System Command from Within a Procedure

Use the SYSTEM statement to execute a system call from a procedure. [Figure 8-11](#) shows a call to the `echo` command.

Figure 8-11
Making a system call from a procedure with a SYSTEM statement

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);

DELETE FROM customer
  WHERE customer_num = cnum;

IF username = "acctclrk" THEN
  SYSTEM "echo 'Delete from customer by acctclrk' >> /mis/records/updates"
;
END IF
END PROCEDURE -- delete_customer
```

Recursively Calling a Procedure

You can call a procedure from itself. There are no restrictions on calling a procedure recursively.

Passing Information into and out of a Procedure

When you create a procedure, you determine whether it expects information to be passed to it by specifying an argument list. For each piece of information that the procedure expects, you specify one argument and the data type of that argument.

For example, if a procedure needs to have a single piece of integer information passed to it, you can provide a procedure heading as follows:

```
CREATE PROCEDURE safe_delete(cnum INT)
```

Returning Results

A procedure that returns one or more values must contain two lines of code to accomplish the transfer of information. You must state the data types that are going to be returned, and you must return the values explicitly.

Specifying Return Values

Immediately after specifying the name and input parameters of your procedure, you must include a RETURNING clause with the data type of each value you expect to be returned. The following example shows the header of a procedure (name, parameters, and RETURNING clause) that expects one integer as input and returns one integer and one 10-byte character value:

```
CREATE PROCEDURE get_call(cnum INT)
RETURNING INT, CHAR(10);
```

Returning the Value

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 your procedure to return the same number and data types as listed in the RETURNING clause. The following example shows how you can return information from the `get_call` procedure:

```
CREATE PROCEDURE get_call(cnum INT)
  RETURNING INT, CHAR(10);
  DEFINE ncalls INT, o_name CHAR(10);
  .
  .
  .
  RETURN ncalls, o_name;
  .
  .
  .
END PROCEDURE
```

Returning More Than One Set of Values from a Procedure

If your procedure performs a select that can return more than one row from the database or if you return values from inside a loop, you must use the WITH RESUME keywords in the RETURN statement. Using a RETURN...WITH RESUME statement causes the value or values to be returned to the calling program or procedure. After the calling program receives the values, execution returns to the statement immediately following the RETURN...WITH RESUME statement.

[Figure 8-12](#) is an example of a cursory procedure. It returns values from a FOREACH loop and a FOR loop. This procedure is called a cursory procedure because it contains a FOREACH loop.

Figure 8-12*Procedure that returns values from a FOREACH loop and a FOR loop*

```

CREATE PROCEDURE read_many (lastname CHAR(15))
  RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15), CHAR(2),
  CHAR(5);

  DEFINE p_lname, p_fname, p_city CHAR(15);
  DEFINE p_add CHAR(20);
  DEFINE p_state CHAR(2);
  DEFINE p_zip CHAR(5);
  DEFINE lcount INT ;
  DEFINE i INT ;

  LET lcount = 0;
  TRACE ON;
  CREATE VIEW myview AS SELECT * FROM customer;
  TRACE "Foreach starts";
  FOREACH
  SELECT fname, lname, address1, city, state, zipcode
         INTO p_fname, p_lname, p_add, p_city, p_state, p_zip
         FROM customer
         WHERE lname = lastname
  RETURN p_fname, p_lname, p_add, p_city, p_state, p_zip WITH
  RESUME;
  LET lcount = lcount +1;
  END FOREACH;

  FOR i IN (1 TO 5)
  BEGIN
    RETURN "a", "b", "c", "d", "e" WITH RESUME;
  END
  END FOR;
END PROCEDURE

```

When you execute this procedure, it returns the name and address for each person with the specified last name. It also returns a sequence of letters. The calling procedure or program must be expecting multiple returned values and it must use a cursor or a FOREACH statement to handle the multiple returned values.

Exception Handling

You can trap any exception (or error) returned by the database server to your procedure, or raised by your procedure, by using the ON EXCEPTION statement. The RAISE EXCEPTION statement enables you to generate an exception within your procedure.

Trapping an Error and Recovering

The ON EXCEPTION statement provides a mechanism to trap any error.

To trap an error, enclose a group of statements in a statement block and precede the statement block with an ON EXCEPTION statement. If an error occurs in the block that follows the ON EXCEPTION statement, recovery action can be taken.

Figure 8-13 shows an example of an ON EXCEPTION statement within a BEGIN...END block.

Figure 8-13

An ON EXCEPTION statement within a BEGIN...END block

```
BEGIN
  DEFINE c INT;
  ON EXCEPTION IN
    (
      -206, -- table does not exist
      -217 -- column does not exist
    ) SET err_num

  IF err_num = -206 THEN
    CREATE TABLE t (c INT);
    INSERT INTO t VALUES (10);
    -- continue after the insert statement
  ELSE
    ALTER TABLE t ADD(d INT);
    LET c = (SELECT d FROM t);
    -- continue after the select statement.
  END IF
  END EXCEPTION WITH RESUME

  INSERT INTO t VALUES (10); -- will fail if t does not exist

  LET c = (SELECT d FROM t); -- will fail if d does not exist
END
```

When an error occurs, the SPL interpreter searches for the innermost ON EXCEPTION declaration that traps the error. Note that the first action after trapping the error is to reset the error. When execution of the error action code is complete, and if the ON EXCEPTION declaration that was raised included the WITH RESUME keywords, execution resumes automatically with the statement *following* the statement that generated the error. If the ON EXCEPTION declaration did not include the WITH RESUME keywords, execution exits the current block completely.

Scope of Control of an ON EXCEPTION Statement

An ON EXCEPTION statement is valid for the statement block that follows the ON EXCEPTION statement, all of the statement blocks nested within that following statement block, and all of the statement blocks that follow the ON EXCEPTION statement. It is *not* valid in the statement block that contains the ON EXCEPTION statement.

The pseudocode in [Figure 8-14](#) shows where the exception is valid within the procedure. That is, if error 201 occurs in any of the indicated blocks, the action labeled a201 occurs.

Figure 8-14

Pseudocode showing the ON EXCEPTION statement is valid within a procedure

```
CREATE PROCEDURE scope()
  DEFINE i INT;
  .
  .
  .
  BEGIN -- begin statement block A
  .
  .
  .
    ON EXCEPTION IN (201)
    -- do action a201
  END EXCEPTION
  BEGIN -- statement block aa
    -- do action, a201 valid here
  END
  BEGIN -- statement block bb
    -- do action, a201 valid here
  END
  WHILE i < 10
    -- do something, a201 is valid here
  END WHILE
```

```
END
BEGIN -- begin statement block B
  -- do something
  -- a201 is NOT valid here
END
END PROCEDURE
```

User-Generated Exceptions

You can generate your own error using the RAISE EXCEPTION statement, as shown in the following pseudocode example. In this example, the ON EXCEPTION statement uses two variables, **esql** and **eisam**, to hold the error numbers returned by the database server. If an error occurs and the SQL error number is -206, the action defined in the IF clause is taken. If any other SQL error is caught, it is passed out of this BEGIN...END block to the block that contains this block.

```
BEGIN
  ON EXCEPTION SET esql, eisam -- trap all errors
  IF esql = -206 THEN          -- table not found
    -- recover somehow
  ELSE
    RAISE exception esql, eisam ; -- pass the error up
  END IF
END EXCEPTION
  -- do something
END
```

Simulating SQL Errors

You can generate errors to simulate SQL errors, as shown in the following example. Here, if the user is **pault**, then the stored procedure acts as if that user has no update privileges, even if he really does have that privilege.

```
BEGIN
  IF user = "pault" THEN
    RAISE EXCEPTION -273; -- deny Paul update privilege
  END IF
END
```


Using RAISE EXCEPTION to Exit Nested Code

You can use the RAISE EXCEPTION statement to break out of a deeply nested block, as shown in [Figure 8-15](#). If the innermost condition is true (if aa is negative), then the exception is raised and execution jumps to the code following the END of the block. In this case, execution jumps to the TRACE statement.

Figure 8-15
Breaking out of nested loop with a RAISE EXCEPTION statement

```

BEGIN
  ON EXCEPTION IN (1)
  END EXCEPTION WITH RESUME -- do nothing significant (cont)

  BEGIN
    FOR i IN (1 TO 1000)
      FOREACH select ..INTO aa FROM t
        IF aa < 0 THEN
          RAISE EXCEPTION 1 ;    -- emergency exit
        END IF
      END FOREACH
    END FOR
  RETURN 1;
END

--do something;                -- emergency exit to
                                -- this statement.

TRACE "Negative value returned"
RETURN -10;
END

```

Remember that a BEGIN...END block is a *single* statement. When an error occurs somewhere inside a block and the trap is outside the block, when execution resumes, the rest of the block is skipped and execution resumes at the next statement.

Unless there is a trap for this error nested somewhere in the block, the error condition is passed back to the block that contains the call and on back to any blocks that contain the block. If there is no ON EXCEPTION statement that is set to handle the error, execution of the procedure stops, creating an error for the program or procedure that is executing the procedure.

SPL Statement Syntax

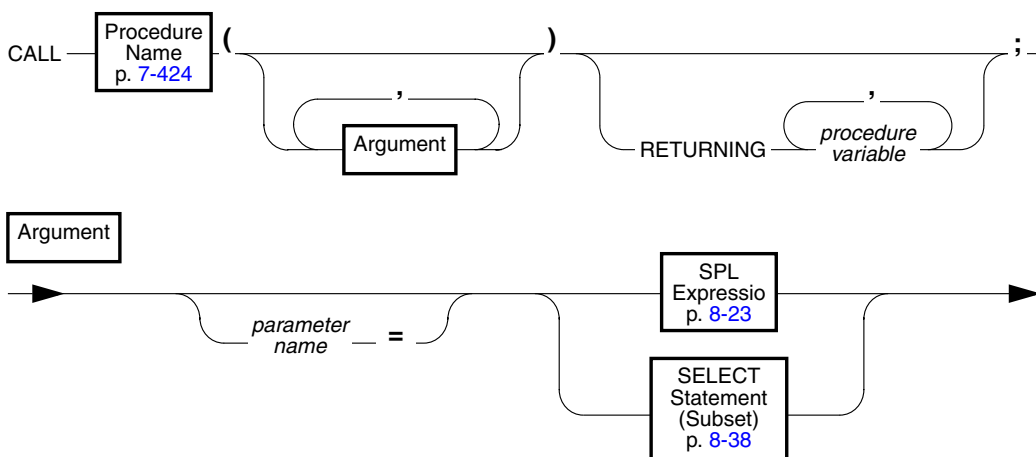
The rest of this chapter describes the statements in SPL. The syntax and usage for each statement are listed, as well as examples. For a description of the symbols used in the syntax diagrams, see [“Syntax Conventions”](#) on page 12 of the Introduction.

CALL

Purpose

Use the CALL statement to execute a procedure from within a stored procedure.

Syntax



parameter name is the name of the parameter as defined by its CREATE PROCEDURE statement.

procedure variable is the name of a variable as defined by its CREATE PROCEDURE statement.

Usage

The CALL statement invokes a procedure called *procedure name*. The CALL statement is identical in behavior to the EXECUTE PROCEDURE statement, except it can be used only from within a stored procedure.

Specifying Arguments

If more arguments are in a CALL statement than are expected by the called procedure, an error is returned.

If fewer arguments are specified by a CALL statement than are expected by the called procedure, the arguments are said to be missing. Missing arguments are initialized to their corresponding default values, if default values were specified. (See CREATE PROCEDURE on [page 7-58](#).) 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.

Procedure arguments are bound to procedure parameters by name or position, but not both. That is, you can use the *parameter name* = syntax for none or all of the arguments specified in one CALL statement.

For example, both of the following procedure calls are valid for a procedure that expects character arguments t, n, and d, in that order:

```
CALL add_col (t="customer", d="integer", n = "newint");  
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, as long as it returns exactly one value of the proper type and length. (See the discussion of SELECT statements that begins on [page 7-258](#) for more information.)

Receiving Input from the Called Procedure

The RETURNING clause specifies the *procedure variables* that receive the returned values from a procedure call. If the RETURNING clause is omitted, the called procedure must not return any values.

The following example shows two procedure calls, one that expects no values to be returned (**no_args**) and one that expects three values to be returned (**yes_args**). Three integer variables have been defined 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
```

References

In this manual, see the EXECUTE PROCEDURE statement.

CONTINUE

Purpose

Use the CONTINUE statement to start the next iteration of the innermost loop of the identified type.

Syntax

```
CONTINUE _____ ;
```

Usage

When a CONTINUE statement is encountered, the rest of the statements in the innermost loop of the indicated type are skipped. Execution continues at the top of the loop with the next iteration. For example, in the following procedure, the values 3 through 15 are inserted into the table **testtable**. The values 3 through 9 and 13 through 15 also are returned in the process. The value 11 is not returned because the CONTINUE FOR statement is reached and causes the RETURN i WITH RESUME statement to be skipped.

```
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;
```

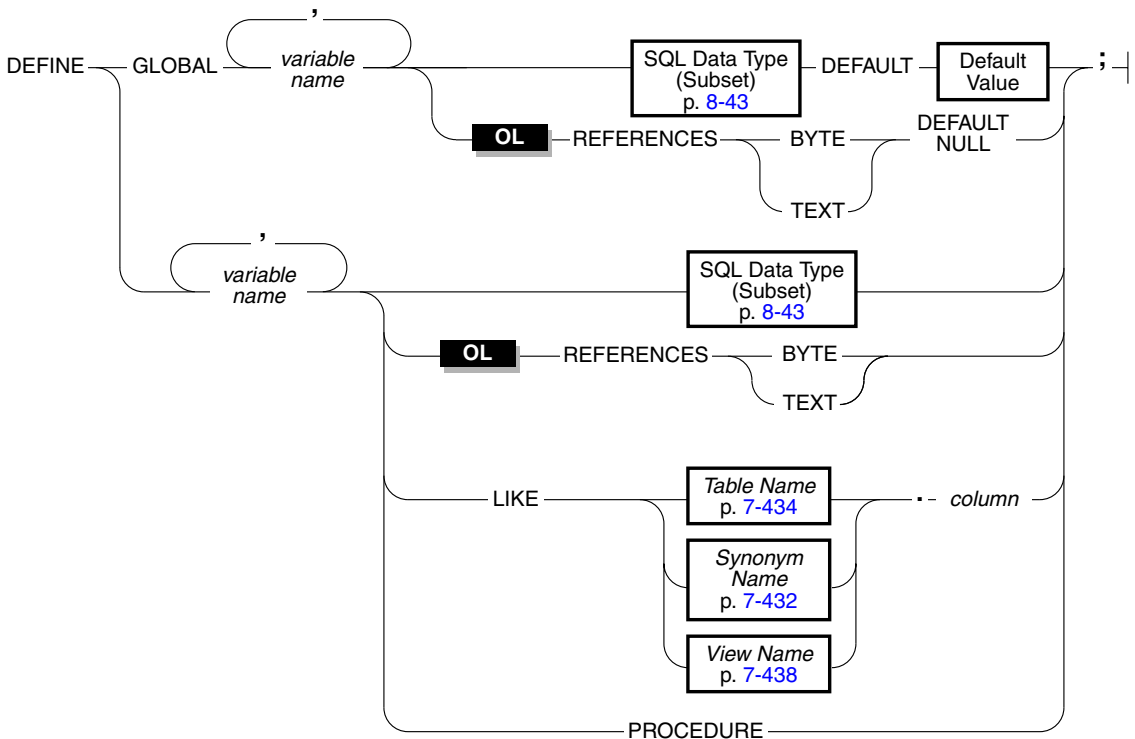
The CONTINUE statement generates errors if it cannot find the identified loop.

DEFINE

Purpose

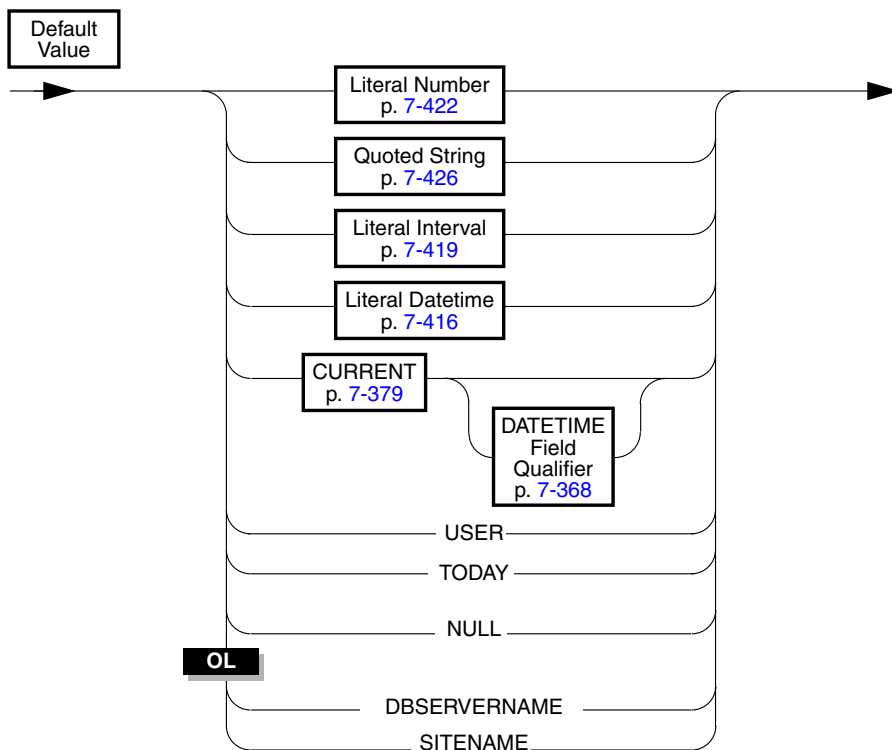
Use the DEFINE statement to declare variables that are used in the procedure and assign them data types.

Syntax



column is the name of a column in the table.

variable name is the name of the procedure variable being defined.



Usage

The DEFINE statement is not an executable statement. The DEFINE statement must appear after the procedure header and before any other statements. A variable can be used anywhere within the statement block in which 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 of the SQL data types except SERIAL, TEXT, and BYTE.

Defining TEXT and BYTE Variables

You can use TEXT and BYTE variables by using the REFERENCES keyword. TEXT and BYTE variables do not contain the actual data but are simply pointers to the data. The REFERENCES keyword is a reminder that the procedure variable is just a pointer. You use the procedure variables for TEXT and BYTE 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. A variable can be redefined 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 following redeclaration is allowed. 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 keyword GLOBAL are available to other procedures. The types of these variables must match the types of variables in the global environment. The global environment is the memory used by all of the procedures run within a given session (a DB-Access session or an embedded-language program session). The values of global variables are stored in memory.

Global variables are shared between procedures running in the current session. Since global variables are not saved in the database, the global variables are lost when the current session is closed.

Global variables are not shared across procedures running in remote database servers. Global procedure variables are not shared between the database server and any application development tools.

The first declaration of a global variable establishes the variable in the global environment; subsequent global declarations simply bind the variable to the global environment, establishing the value of the variable at that point. Consider the following example of two procedures, **proc1** and **proc2**, which have both defined the global variable **gl_out** as follows:

```
CREATE PROCEDURE proc1()
.
.
.
DEFINE GLOBAL gl_out INT DEFAULT 13;
.
.
.
LET gl_out = gl_out + 1;
END PROCEDURE;

CREATE PROCEDURE proc2()
.
.
.
DEFINE GLOBAL gl_out INT DEFAULT 23;
DEFINE tmp INT;
.
.
.
LET tmp = gl_out
.
.
.
END PROCEDURE;
```

If **proc1** is called first, **gl_out** is set to 13 and then incremented to 14. If **proc2** is then called, it sees that the value of **gl_out** is already defined, so the default value of 23 is not applied. Then, **proc2** assigns the existing value of 14 to **tmp**. If **proc2** had been called first, **gl_out** would have been set to 23, and 23 would have been assigned to **tmp**. Later calls to **proc1** would not apply the default of 13.

Providing Default Values

You can provide a literal value or a null value as the default for a global variable. You also can use a call to an SQL function to provide the default value. The following procedure 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 or VARCHAR value of at least 18 characters.

USER

If you use USER as the default, the variable must be a CHAR or VARCHAR value of at least 8 characters.

CURRENT

If you use `CURRENT` as the default, the variable must be a `DATETIME` value. If your variable has been qualified with the keywords `YEAR TO FRACTION`, 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. For example, the following `DEFINE` statement defines a `DATETIME` variable with qualifiers and uses `CURRENT` with matching qualifiers.

```
DEFINE GLOBAL d_var DATETIME YEAR TO MONTH
        DEFAULT CURRENT YEAR TO MONTH;
```

TODAY

If you use `TODAY` as the default, the variable must be a `DATE` value.

TEXT and BYTE

The only default value possible for a `TEXT` or `BYTE` variable is null. For example, the following procedure defines a global variable call `l_blob` of type `TEXT`:

```
CREATE PROCEDURE use_text()
    DEFINE i INT;
    DEFINE GLOBAL l_blob 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, *variable name* is defined as the same type as the type of the *column* in *table*. The types of variables defined like database columns are resolved at run time; therefore, *column* and *table* need not exist at compile time.

Declaring Variables as the PROCEDURE Type

The PROCEDURE type indicates that in the current scope, *variable name* 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 procedure, not the SQL LENGTH function. This 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** prior to defining and using it in another procedure, as follows:

```
DEFINE length PROCEDURE;  
.  
.  
.  
LET x = length (a,b,c)
```

If you create a procedure named like an aggregate function (SUM, MAX, MIN, AVG, COUNT) or one named **extend**, you must qualify the procedure name with the owner name.

Declaring Variables for BYTE and TEXT Data

The keyword REFERENCES indicates that *variable name* is not a BYTE or TEXT value, but rather, a pointer to the BYTE or TEXT value. You use the variable as though it holds the data itself.

The following example defines a local BYTE variable:

```
CREATE PROCEDURE use_blob()  
  DEFINE i INT;  
  DEFINE l_blob REFERENCES BYTE;  
END PROCEDURE --use_blob
```

If you pass a variable of type TEXT or BYTE to a procedure, the data is passed to the database server and stored in the root dbspace. You do not need to know the location or name of the file that holds the data; only the name of the BYTE or TEXT variable as it is defined in the procedure is needed for BYTE or TEXT manipulation.

EXIT

Purpose

Use the EXIT statement to stop the execution of a FOR, FOREACH, or WHILE loop.

Syntax

```
EXIT _____ ;
```

Usage

The EXIT statement causes the innermost loop of the indicated type (WHILE, FOR, or FOREACH) to terminate. Execution resumes at the first statement outside of the loop.

If the EXIT statement cannot find the identified loop, it fails.

Used outside of all loops, the EXIT statement generates errors.

In the following example, an EXIT FOR statement is used. In the FOR loop, when *j* becomes 6, the IF condition *i* = 5 in the WHILE loop is true. Execution of the FOR loop stops, and execution continues at the next statement outside of the FOR loop, in this case, the END PROCEDURE statement. Thus, in this example, the procedure finishes 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;
```



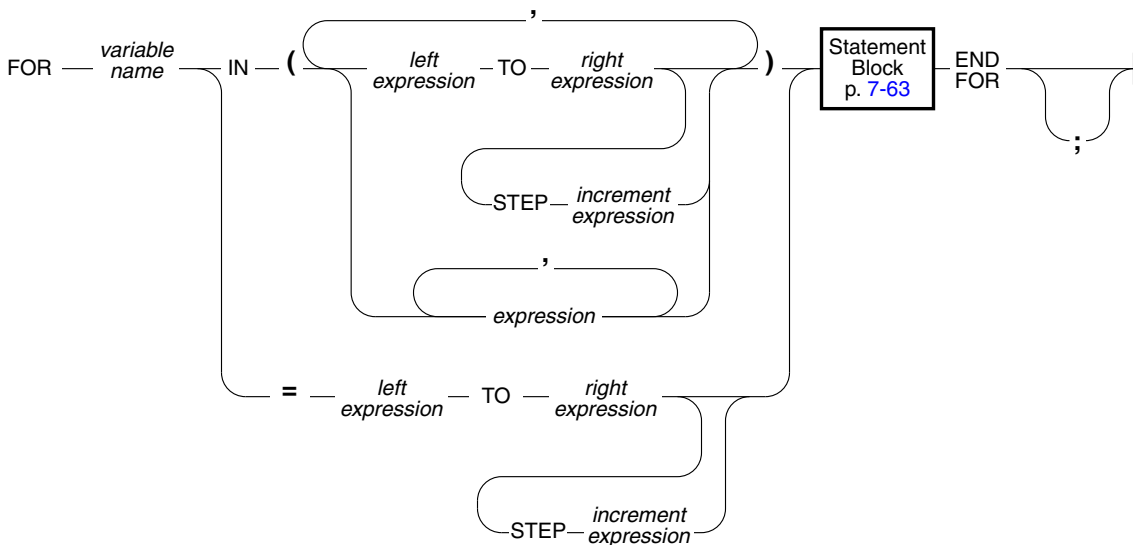
```
        IF i = 5 THEN  
            EXIT FOR;  
        END IF  
    END WHILE  
END FOR  
END PROCEDURE
```

FOR

Purpose

Use the FOR statement to initiate a controlled (definite) loop in cases where 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



expression is a numeric or character value. The data type of *expression* must match the data type of the *variable name*. You can use the output of a SELECT statement as an *expression*.

increment is a positive or negative amount by which *variable name*

expression is to be incremented. The increment expression cannot evaluate to zero.

left expression is the starting expression of a range. The left expression must match the data type of the *variable name*.

right expression is the ending expression in the range.

variable name is a variable that is already defined and valid within this statement block.

Usage

All expressions are computed before the execution of the FOR statement begins. If one or more of the expressions are variables and their values are changed during the loop, the change has no effect on the iterations of the loop.

The FOR loop terminates when *variable name* takes on the values of each element in the expression list or range in succession or when an EXIT FOR statement is encountered.

An error is generated if an assignment within the body of the FOR statement attempts to modify the value of *variable name*.

Using the TO Keyword to Define a Range

The TO keyword implies a range operator; the range is defined by *left expression* and *right expression*, and the number of increments is set implicitly with the STEP *increment expression* option. If you use the TO keyword, the *variable* must be an INT or SMALLINT data type. The following example shows two equivalent FOR statements. Both use the TO keyword to define a range. The first statement uses the IN keyword, while the second statement uses an equal sign (=). Both statements cause 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, *increment expression* is given the value of -1 if *right expression* is less than *left expression*, or +1 if *right expression* is more than *left expression*. If the *increment expression* is specified, it must be negative if *right expression* is less than *left expression*, or positive if *right expression* is more than *left expression*. The two statements in the following example are equivalent. In the first statement, the STEP increment is explicit. In the second statement, the STEP increment is implicitly 1.

```
FOR index IN (12 to 21 STEP 1)
  -- statement block
END FOR

FOR index = 12 TO 21
  -- statement block
END FOR
```

The value of *variable name* is initialized to the value of *left expression*. In subsequent iterations, *increment expression* is added to the value of *variable name* and checked to determine whether the value of *variable name* still is between *left expression* and *right expression*. If so, then the next iteration occurs; otherwise, the loop is exited or, if there is another range specified, 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 using 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 value of *variable name* is initialized to the value of the first *expression* specified. In subsequent iterations, *variable name* takes on the value of the next *expression*. When the last expression in the list is used, the loop stops.

The expressions in the IN list are not limited to numeric values, as long as no range operators are used in the IN list. The following example uses a character expression list:

```
FOR c IN ("hello", (SELECT name FROM t), "world", v1, v2)
  INSERT INTO t VALUES (c);
END FOR
```

The following FOR statement shows the use of a numeric expression list:

```
FOR index IN (15,16,17,18,19,20,21)
  -- statement block
END FOR
```

Mixing Range and Expression Lists in the Same FOR Statement

If *variable name* is an INT or SMALLINT value, you can mix ranges and expression lists in the same FOR statement. The following example shows a mixture using an integer variable. Values in the expression list include the value returned from a SELECT statement, a sum of an integer variable and a constant, the values returned from a procedure named `p_get_int`, and integer constants.

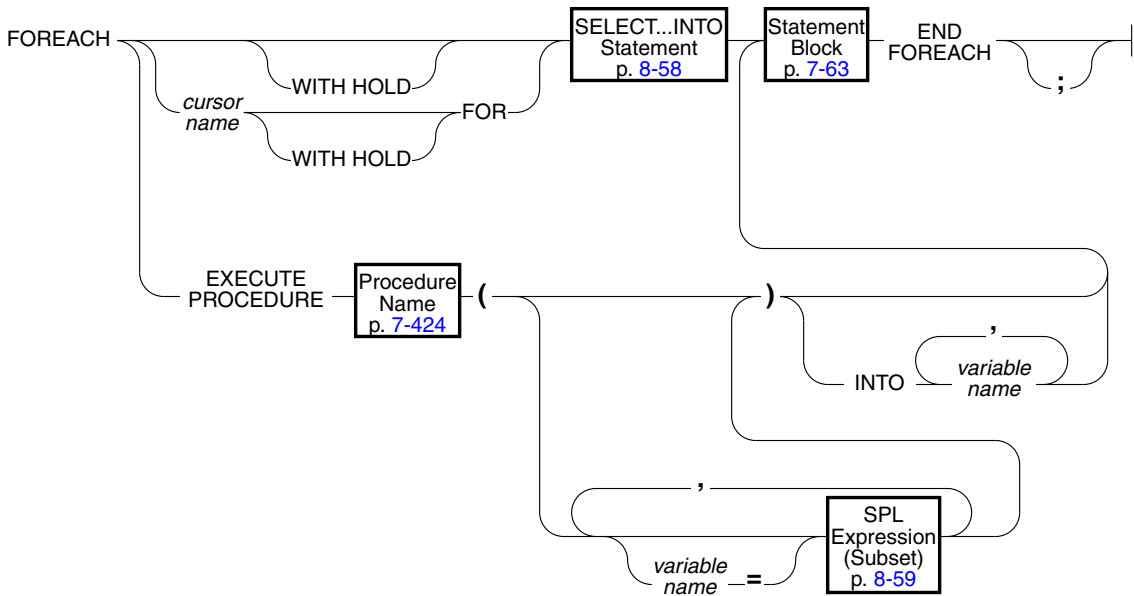
```
CREATE PROCEDURE for_ex ()
  DEFINE i, j INT;
  LET j = 10;
  FOR i IN (1 TO 20, (SELECT c1 FROM tab WHERE id = 1),
  j+20 to j-20, p_get_int(99),98,90 to 80 step -2)
    INSERT INTO tab VALUES (i);
  END FOR
END PROCEDURE
```

FOREACH

Purpose

Use a FOREACH loop to select and manipulate more than one row.

Syntax



cursor name is an identifier that you supply as a name for the SELECT... INTO statement.

variable name is the name of a procedure variable.

Usage

A FOREACH loop is the procedural equivalent of using a cursor. When a FOREACH statement is executed, the database server takes the following actions:

1. A cursor is declared and opened implicitly.
2. The first row is obtained from the query contained within the FOREACH loop, or the first set of values is obtained from the called procedure.
3. Each variable in the variable list is assigned the value of the corresponding value from the active set created by the SELECT statement or the called procedure.
4. The statement block is executed.
5. The next row is fetched from the SELECT statement or called procedure on each iteration; step 3 is repeated.
6. The loop terminates when no more rows are found that satisfy the SELECT statement or called procedure. The implicit cursor is closed when the loop terminates.

Since the statement block can contain additional FOREACH statements, cursors can be nested. There is no limit to the number of cursors that can be nested.

A procedure that returns more than one row or set of values is called a *cursor 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:

- No further rows are returned by the cursor.
- The cursor is a select cursor without a HOLD specification and a transaction is completed using COMMIT or ROLLBACK statements.
- An EXIT statement is executed that 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 8-67.)
- A cursor in the calling procedure that is executing this cursory procedure (within a FOREACH loop) is closed for any reason.

Using a **SELECT...INTO** Statement

The SELECT statement in the FOREACH statement must include the INTO clause. It also can include UNION and ORDER BY clauses. It cannot use the INTO TEMP clause. The syntax of a SELECT statement is shown on page 7-258.

The type and count of each variable in the variable list must match each value returned by the SELECT...INTO statement.

Hold Cursors

Using the WITH HOLD keywords specifies that the cursor should remain open when a transaction is closed (committed or rolled back).

Updating or Deleting Rows Identified by Cursor Name.

To update or delete the current row of *cursor name*, use the WHERE CURRENT OF *cursor name* clause.

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 returned by the called procedure.

Subset of Expressions Allowed in the Procedure Parameters

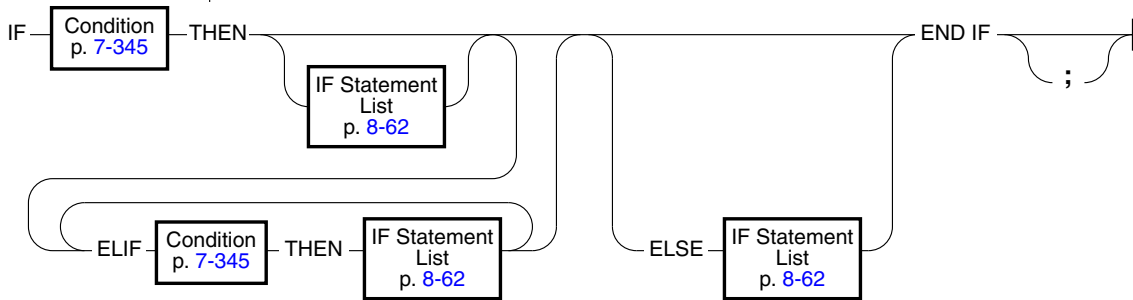
You can use any SPL expression as a procedure parameter. If you use a subquery or procedure call, the subquery or procedure must return a single value of the appropriate type and size. For the full syntax of an SPL expression, see page [8-23](#).

IF

Purpose

Use an IF statement to create a branch within a procedure.

Syntax



Usage

The condition stated in the IF clause is evaluated. If the result is true, then the statements following the THEN keyword are executed. If the result is false and there is an ELIF clause, the statements following the ELIF clause are executed. If there is no ELIF clause, or if the condition in the ELIF clause is not true, the statements following the ELIF keywords are executed.

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 zero 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

```

The ELIF Clause

Use the ELIF clause to specify one or more additional conditions to evaluate.

If you specify an ELIF clause and the IF condition is false, the ELIF condition is evaluated. If the ELIF condition is true, the statements following the ELIF clause are executed.

The ELSE Clause

The ELSE clause is executed if no true previous condition is in the IF clause or any of the ELIF clauses.

Conditions in an IF Statement

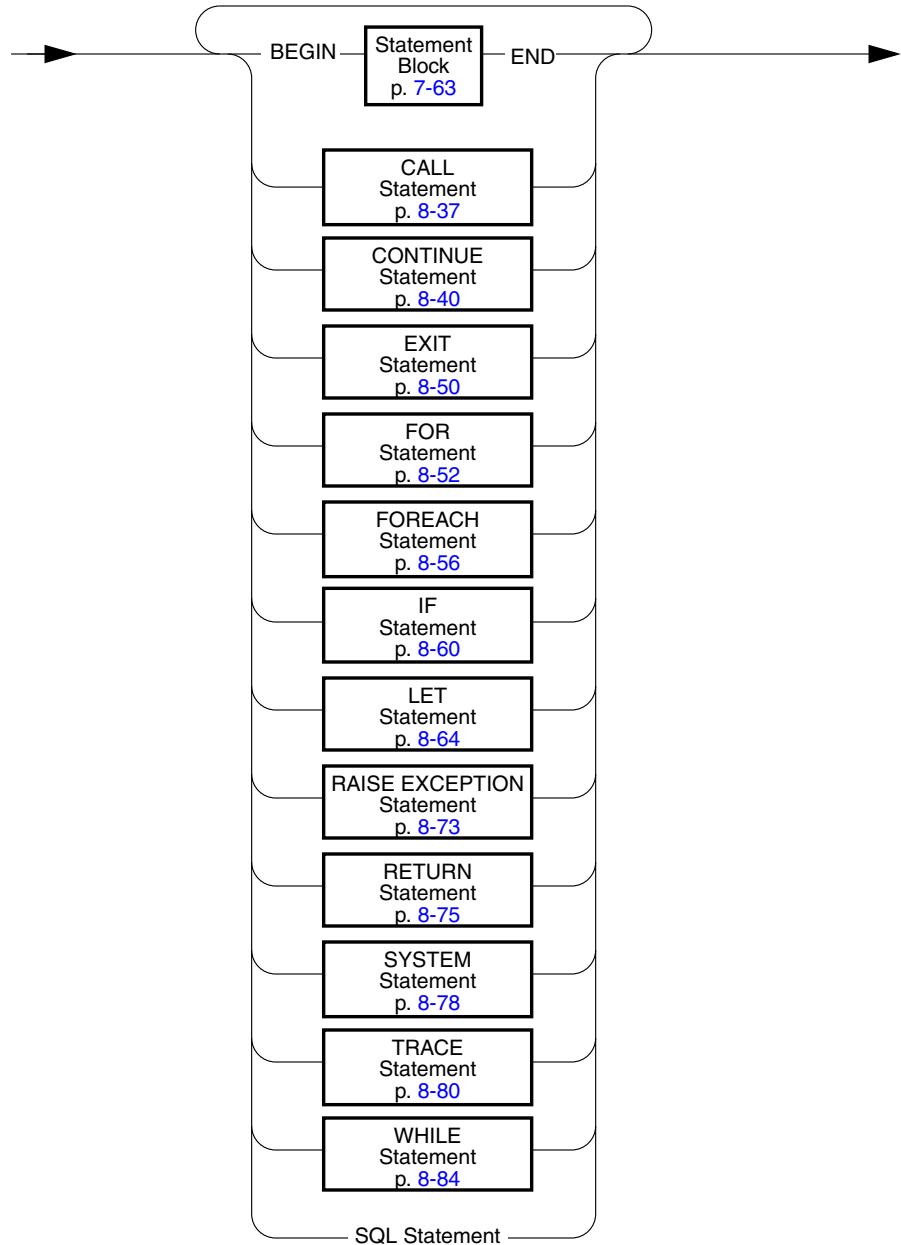
Conditions in an IF statement are evaluated in the same way as conditions in a WHILE statement.

If any expression contained within the condition evaluates to null, then the condition automatically becomes not true. In particular:

1. Let the expression *x* evaluate to null. Then *x* is not true by definition. Furthermore, not (*x*) is also *not* true.
2. The only operator that can yield true for *x* is the IS NULL operator. That is, *x* IS NULL is true and *x* IS NOT NULL is not true.

If an expression within the condition has an UNKNOWN value (due to the use of an uninitialized variable), it is an immediate error. The statement terminates and an exception is raised.

IF Statement List



Subset of SQL Statements Allowed in an IF Statement

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

- CHECK TABLE
- CLOSE DATABASE
- CREATE DATABASE
- CREATE PROCEDURE
- DATABASE
- INFO
- LOAD
- OUTPUT
- REPAIR TABLE
- ROLLFORWARD DATABASE
- START DATABASE
- UNLOAD

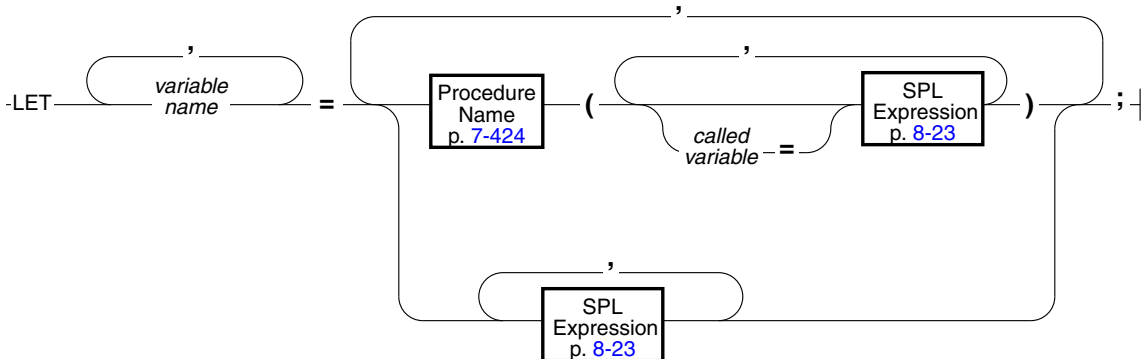
You can use a SELECT statement only if you use the INTO TEMP clause to put the results of the SELECT statement into a temporary table.

LET

Purpose

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



called variable is a procedure variable of the called procedure.

variable name is a procedure variable.

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 the type of *variable name*, if possible, and the assignment takes place. If conversion is not possible, an error is generated and the value of *variable name* is undefined.

A compound assignment assigns multiple expressions to multiple variables. The count and type of *expressions* in the expression list must match the count and 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 not legal:

```
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-variable* statement in a procedure. You can use a SELECT statement to assign values to one or more variables on the left-hand side of the = operator. The following examples use a SELECT statement in a LET statement:

```
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. For example, the following code is not legal:

```
LET a,b = (SELECT c1,c2 FROM t) + (10,15); -- ILLEGAL CODE
```

Because a LET statement is equivalent to a SELECT...INTO statement, the following two statements in this procedure 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, the SELECT statement must be enclosed 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 of the necessary arguments to the procedure in the LET statement, unless the procedure has default values for its arguments.

If you use the *variable name* = syntax for one of the parameters in the called procedure, you must use it for all of the parameters.

The *variable name* receives the returned value from a procedure call. A procedure can return more than one value into a list of *variable names*. A procedure that returns more than one row must be enclosed in a FOREACH loop.

The following example shows two valid LET statements that contain procedure calls. The third LET statement is not legal, because it tries to have the output of two procedures added and then assigned to two variables, a and b. This LET statement can be split easily into two legal LET statements:

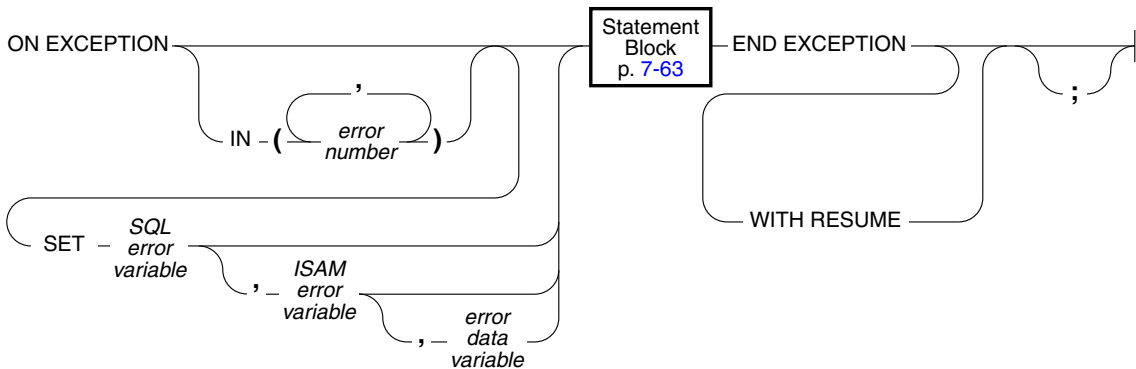
```
LET a, b, c = proc1(name = "grok", age = 17);  
LET a, b, c = 7, proc ('orange', 'green');  
LET a, b = proc1() + proc2(); -- ILLEGAL CODE
```


ON EXCEPTION

Purpose

Use the ON EXCEPTION statement to specify the actions that are taken for a particular error or a set of errors.

Syntax



error data variable is a literal string or a variable that contains a string returned by an SQL error.

error number is an SQL error number, or an error number created by a RAISE EXCEPTION statement, that is to be trapped.

ISAM error variable is an integer variable that receives the ISAM error number of the exception raised.

SQL error variable is an integer variable that receives the SQL error number of the exception raised.

Usage

The ON EXCEPTION statement, together with the RAISE EXCEPTION statement, provides an error-trapping and recovery mechanism for SPL. The ON EXCEPTION statement defines a list of errors that are to be trapped as the 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 of the statement blocks nested within that following statement block, and all of the statement blocks that follow the ON EXCEPTION statement.

The exceptions 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 and there is no accompanying ISAM error, a zero is returned to the variable. If you specify a variable to receive the returned error text and there is none, an empty string is put into the variable.

Placement of the ON EXCEPTION Statement

The ON EXCEPTION statement 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. The ON EXCEPTION statement is used after the DEFINE statement and before the body of the procedure. This 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 takes place.

```
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 that traps the particular error code. This can be an ON EXCEPTION statement that has the error number in the IN clause, or an ON EXCEPTION statement without an IN clause. If no pertinent ON EXCEPTION statement is found, the error code is passed back to the caller (procedure, application, or interactive user), and execution aborts.

The following example uses two ON EXCEPTION statements with the same error number so that error number 691 can be trapped in two different 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 a default trapping situation. For example, the sequence of statements in the following example has a net effect of saying, "Test for an error. If it is error -210, -211, or -212, take action A. If it is error -300, take action B. If it is any other error, take action C."

```
CREATE PROCEDURE ex_test ()
.
.
.
ON EXCEPTION
SET error_num
-- action C
END EXCEPTION

ON EXCEPTION IN (-300)
-- action B
END EXCEPTION
ON EXCEPTION IN (-210, -211, -212)
SET error_num
-- action A
END EXCEPTION
.
.
.
```

Receiving Error Information in the SET Clause

If you use the SET clause, when an exception occurs, the SQL error number and (optionally) the ISAM code are inserted into the variables specified in the SET clause. If you provided an *error data variable*, any error text returned by the database server is put into the *error data variable*. Error text includes such information as the offending table or column name.

Forcing Continuation of the Procedure with the WITH RESUME Keywords

The example on page 8-69 uses the WITH RESUME keywords to indicate that after the statement block in the ON EXCEPTION statement is executed, 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, this means that the count of salespeople names occurs even if the error occurred.

Continuation of Procedure Execution After an Exception Occurs

If you do not include the WITH RESUME keywords in your ON EXCEPTION statement, after an exception is raised the next statement executed depends on the placement of the ON EXCEPTION statement, as follows:

- If the ON EXCEPTION statement is inside a statement block with a BEGIN and an END keyword, then 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), then the rest of the loop is skipped and execution resumes with the next iteration of the loop.
- If the ON EXCEPTION statement is not contained within any statement or block but only in the procedure itself, then the procedure terminates by executing a RETURN statement with no arguments. 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

Purpose

Use the RAISE EXCEPTION statement to simulate the generation of an error.

Syntax

```
RAISE EXCEPTION SQL error ;
                , ISAM error
                , error text
```

error text is a quoted string or variable that contains a string to be returned by the SQL error.

ISAM error is an SPL expression that evaluates to an integer value that is a valid ISAM error number.

SQL error is an SPL expression that evaluates to an integer value that is a valid SQL error number.

Usage

The RAISE EXCEPTION statement is used to simulate an error. The generated error can be trapped by an ON EXCEPTION statement.

If the *ISAM error* expression is omitted, the ISAM error code is set to zero when the exception is raised. (If you want to use the error text field but not specify the ISAM error number portion, you can *ISAM error* be zero.) For example, the following statement raises the error number 99999 and returns the stated text:

```
RAISE EXCEPTION -99999, 0, "You broke the rules";
```

The exceptions raised can be either system- or user-generated.

In the following example, if the value of a is negative, exception 99999 is raised. An ON EXCEPTION statement that traps for an exception of 99999 should be somewhere in the code.

```
FOREACH SELECT c1 INTO a FROM t
  IF a < 0 THEN
    RAISE EXCEPTION 99999-- emergency exit
  END IF
END FOREACH
```

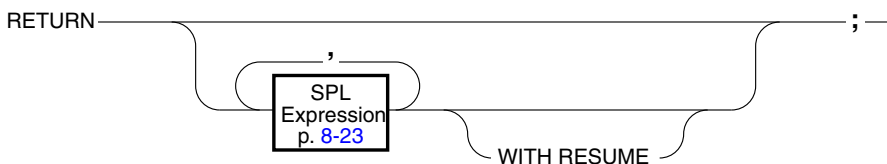
See the ON EXCEPTION statement for more information about scope and compatibility of exceptions.

RETURN

Purpose

Use the RETURN statement to designate the values that are returned by the procedure 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 that defined the procedure. The number and type of values in the RETURN statement, if any, must match in number and type the types listed in the RETURNING clause of the CREATE PROCEDURE statement. You can choose to return no values even if you specified 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, returning no value.

The WITH RESUME Keywords

If you use the WITH RESUME keywords after the RETURN statement executes, the next invocation of this procedure (upon the next FETCH or FOREACH statement) starts from the statement following the RETURN statement. If a procedure executes a RETURN WITH RESUME statement, it must be called from a FOREACH loop in the calling procedure or program.

ESQL

If a procedure executes a RETURN WITH RESUME statement, it can be called with a FETCH statement in an application written in an embedded language. ♦

The following example shows a cursory procedure that can be called by another procedure. After the RETURN i WITH RESUME statement returns each value to the calling procedure, the next time **sequence** is called, the next line of **sequence** is executed. 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
```


In both DBA-privileged and owner-privileged procedures that contain SYSTEM statements, the operating system command is run with the permissions of the user executing the procedure.

The following example shows the use of a SYSTEM statement:

```
CREATE PROCEDURE sensitive_update()
.
.
.
LET mailcall = "mail headhoncho < alert"
-- code that evaluates if operator tries to execute a
  certain
-- command, then send email to system administrator
SYSTEM mailcall
.
.
.
END PROCEDURE -- sensitive_update
```

As with other statements, you can use a double pipe symbol (||) to concatenate expressions together with a SYSTEM statement, as follows:

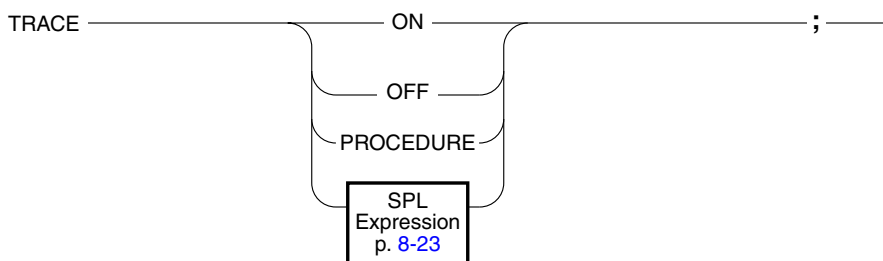
```
CREATE PROCEDURE sensitive_update2()
.
.
.
-- code that evaluates if operator tries to execute a
  certain
-- command, then send email to system administrator
SYSTEM "mail -s violation" ||user1 || " " ||
      user2 || " < violation_file"
.
.
.
END PROCEDURE
```

TRACE

Purpose

Use the TRACE statement to control the generation of debugging output.

Syntax



Usage

Using the TRACE statement generates output that is sent to the file specified by the SET DEBUG FILE statement.

Tracing prints out the current values of all the following items:

- Variables
- Procedure arguments
- Return values
- SQL error codes
- ISAM error codes

The output of each executed TRACE statement is on a separate line.

If you use the TRACE statement without first specifying a DEBUG file to contain the output, an error is generated.

STAR

INET

The trace *state* is inherited by called procedures. 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.

The trace state is not inherited by a procedure that is executed in a remote database server. ♦

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. Turning 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 are traced, not the body of the procedure.

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 being 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 have established a trace-output file using the SET DEBUG statement.

The following example uses the TRACE statement with an expression:

```

CREATE PROCEDURE tracing ()
  DEFINE i INT;
BEGIN
  ON EXCEPTION IN (1)
  END EXCEPTION; -- do nothing
  TRACE OFF;
  SET DEBUG FILE TO "/tmp/foo.trace";
  TRACE "Forloop starts";
  FOR i IN (1 TO 1000)
    BEGIN
      TRACE "FOREACH starts";
      FOREACH SELECT...INTO a FROM t
        IF <some condition> THEN
          RAISE EXCEPTION 1 -- emergency exit
        END IF
      END FOREACH

      -- return some value
    END
  END FOR

  -- do something
END;
END PROCEDURE

```

The following example shows additional TRACE statements:

```

CREATE PROCEDURE testproc()
  DEFINE i INT;

  TRACE OFF;
  SET DEBUG FILE TO "/tmp/test.trace";
  TRACE 'Entering foo';

  TRACE PROCEDURE;
  LET i = testtoo();

  TRACE ON;
  LET i = i + 1;

  TRACE OFF;
  TRACE 'i+1 = ' || i+1;
  TRACE 'Exiting testproc';

  SET DEBUG FILE TO "/tmp/test2.trace";

END PROCEDURE;

```


Looking at the Traced Output

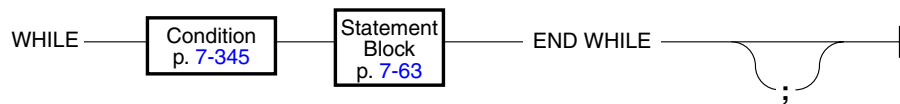
To see the traced output, use an editor or utility to display or read the contents of the file.

WHILE

Purpose

Use the WHILE statement to establish an indefinite loop within a procedure.

Syntax



Usage

The condition is evaluated once at the beginning of the loop. Subsequently, the condition is evaluated 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 contained 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, it is an immediate error. In this case, the loop terminates and an exception is raised.

```

CREATE PROCEDURE simp_while()
  DEFINE i INT;
  DEFINE pf_name CHAR(15);
  WHILE EXISTS (SELECT fname INTO pf_name FROM customer
    WHERE customer_num > 400)
    DELETE FROM customer WHERE id_2 = 2;
  END WHILE

  LET i = 1;
  WHILE i < 10
    INSERT INTO tab_2 VALUES (i);
    LET i = i + 1;
  END WHILE;
END PROCEDURE;

```

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Glossary

abort	To interrupt an active process before completion. The status of data may not be the same as it was before the process began.
access method	A procedure used to retrieve rows from or insert rows into a storage location. In the SET EXPLAIN statement, access method refers to the type of table access in a query, for example, SEQUENTIAL SCAN as opposed to INDEX PATH.
access mode	The status of an open file that determines read and write access to that file.
access privileges	The types of activities that a database user has permission to perform in a specific database, table, or column. Informix database servers maintain their own set of database access privileges that are independent of the operating system access privileges for system files.
active set	The collection of rows satisfying a query associated with a cursor.
aggregate functions	The functions that return a single value based on the values of columns in one or more rows of a table. Examples are the total number, sum, average, maximum, or minimum of an expression in a query or report. Aggregate functions sometimes are referred to as “set functions” or “math functions.”
alias	A temporary alternative name for a table in a query. Usually used in complex subqueries and required for self-joins. In a form-specification file or any SQL query, <i>alias</i> refers to a single-word alternative name used in place of a more complex table name (for example, <i>owner.table name</i>).

ANSI	Acronym for the American National Standards Institute. ANSI sets standards for the computer industry, including standards for SQL languages.
ANSI-compliant	Refers to a database that conforms to certain ANSI performance standards. Informix databases can be created either as ANSI-compliant or not ANSI-compliant. An ANSI-compliant database enforces certain ANSI requirements that are not enforced in databases that are not ANSI-compliant, such as implicit transactions, required owner-naming, and no buffered logging (when using IBM Informix OnLine).
application development tool	Refers to software such as IBM Informix SQL, IBM Informix 4GL, and IBM Informix ESQL that a developer can use to create and maintain a database. The software allows a user to send instructions and data to and receive information from the database server. An application development tool sometimes is referred to as the “front end.”
application productivity tools	Tools, such as forms and reports, used to write applications.
application program	A file or logically related set of files that perform one or more database management tasks.
application tool process	A process that provides the user interface and communicates with either a database server or an IBM Informix NET product.
archiving	Refers to copying all the data and indexes of a database onto a new medium, usually a tape or a different physical device from the one that stores the database.
argument	A value passed to a function, routine, or command.
array	A set of items of the same type. Individual members of the array are referred to as <i>elements</i> and are usually distinguished by an integer argument that gives the position of the element in the array. Informix arrays can have up to three dimensions.
ASCII	Acronym for the American Standards Committee for Information Interchange. Often used to describe an ordered set of printable and non-printable characters used in computers and telecommunication.
attribute	The qualifier for the method of displaying or verifying data in fields of a screen form.
audit trail	A history of all changes to a table in IBM Informix SE database servers.

audit trail log	A file containing a history of all changes to a table. Starting from an archived database, an audit trail log can reconstruct all subsequent changes to the table in IBM Informix SE database servers.
B+ tree	A method of organizing an index for efficient retrieval of records.
backup	To make a duplicate of a computer file on another device or tape to preserve existing work, in case of a computer failure or other mishap.
before-image	The image of a row, page, or other item before any changes are made to it.
blob descriptor	The 56-byte pointer that is stored in a blob column position in a row that includes blob data. The descriptor points to the first segment of the dbspace or blobspace blobpage where the blob data is stored.
blobs	An acronym for Binary Large Objects, blobs are data objects that effectively have no maximum size (theoretically as large as 2^{31} bytes).
blobpage	The unit of disk allocation within a blobspace in IBM Informix OnLine. The System Administrator determines the size of a blobpage; the size can vary from blobpage to blobpage.
blobspace	A logical collection of chunks used to store TEXT and BYTE data in IBM Informix OnLine.
Boolean	A variable or an expression that can take on the logical values TRUE (1), FALSE (0), or UNKNOWN (if null values are involved).
breakpoint	A named object, specified by a debugger, that the programmer can associate with a statement, program block, variable, or logical condition. When a breakpoint is reached, program execution is suspended, allowing the programmer to examine the current value of program variables or the execution stack and to optionally execute debugger commands at that point. A breakpoint must be enabled to take effect. See <i>debugger</i> .
buffer	A portion of computer memory where a program temporarily stores data. Data typically is read into or written out from buffers to disk.
buffered log	Holds transactions in a memory buffer until the buffer is full, regardless of when the transaction is committed or rolled back. IBM Informix OnLine provides this option to speed operations by reducing the number of disk writes.
byte	A unit of storage, approximately corresponding to one character. A kilobyte is 1024 bytes. A megabyte is 10^6 bytes. (When BYTE appears in uppercase letters, it refers to the Informix data type.)

capability	Codes used in a termcap file or terminfo directory that specify terminal functions, such as the size of the screen or whether to clear the screen.
Cartesian product	The set that results when you pair each and every member of one set with each and every member of another set. A Cartesian product results from a multiple-table query when you do not specify the joining conditions among tables. See <i>join</i> .
case sensitivity	The condition of distinguishing between upper- and lowercase letters. Care should be taken in running IBM Informix programs; certain commands and their options are case sensitive, that is, they react differently to the same letters presented in uppercase and lowercase letters.
check constraint	A condition that must be met before data can be assigned to a table column during an INSERT or UPDATE statement. Check constraints are defined using search conditions.
checkpoint	A point in time during IBM Informix OnLine operation when the pages on disk are synchronized with the pages in the shared-memory buffer pool. Checkpoints are marked by a special record written into the logical log.
chunk	A large continuous section of disk space for IBM Informix OnLine. A chunk can correspond to a UNIX disk partition. A specified set of chunks defines a dbspace or blobspace.
client	A computer on a network that uses the resources of a file server or database server.
client/server architecture	Hardware and software design that allows the user interface and database server to reside on separate nodes or platforms on a network.
client/server processing	A function that allows an application to run on multiple computers across a network using the processing power of the computers involved.
close a cursor	To drop the association between the active set of rows resulting from a query and a cursor.
close a database	To relinquish the “current” status of a database. Only one current database can exist at a time.
close a file	To release the association between a file and a program.
cluster an index	To rearrange the physical data of a table according to a specific index.

column	A data element containing a particular type of information that occurs in every row of the table. Column also can refer to a position on the screen or in a window.
command file	A system file containing one or more statements or commands, such as SQL statements or a sequence of programming code.
comment	Information in a program file, report specification, or screen form that is not processed by the computer but that documents the program. You use special characters, such as a pound sign (#), curly braces ({ }), or an asterisk (*) to identify comments.
COMMIT WORK	To terminate a transaction by accepting all changes to the database since the beginning of the transaction.
COMMITTED READ	A level of process isolation available through IBM Informix OnLine in which a user can view only rows that are currently committed at the moment of the query request; that is, a user cannot view rows that were changed as a part of a currently uncommitted transaction. It is the default level of isolation under IBM Informix OnLine for databases that are not ANSI-compliant. See <i>process isolation</i> .
compile	To translate a file containing instructions (in a higher level language) into a file containing the corresponding machine-level instructions.
compile-time errors	Errors that occur at the time the program source code is converted to executable form. See <i>run-time errors</i> .
composite index	An index constructed on two or more columns of a table. The ordering imposed by the composite index varies least frequently on the first named column and most frequently on the last named column.
composite join	A join between two tables based on the relationship among two or more columns in each table.
concatenate	To form the character string that results when a second string is appended to the end of a first string.
concatenation operator	A symbolic notation composed of two pipe symbols () used in expressions to indicate the joining of two strings.
concurrency	The ability of two or more processes to access the same database simultaneously.

configuration file	A file that IBM Informix OnLine uses on initialization that contains configuration data. The configuration file is specified as follows: \$INFORMIXDIR/etc/\$TBCONFIG.
constant	A non-varying data element or value.
constraint	Places restrictions on what kinds of data can be inserted or updated in tables. See also <i>check constraint</i> , <i>primary key constraint</i> , <i>referential constraint</i> , and <i>unique constraint</i> .
control character	A character whose occurrence in a particular context initiates, modifies, or stops a control function (an operation to control a device, for example, in moving a cursor or in reading data). In a program, you can define actions that use the CTRL key in conjunction with another key to execute some programming action (for example, entering CTRL-W to obtain on-line help in IBM Informix products). A control character is sometimes referred to as a "control key." See <i>printable character</i> .
corrupted database	A database whose tables or indexes contain incomplete or invalid data.
corrupted index	An index that does not correspond exactly to its table.
cursor	An identifier associated with a group of rows. Conceptually, the pointer to the current row. You can use cursors for SELECT statements (associating the cursor with the rows returned by a query) or INSERT statements (associating the cursor with a buffer to insert multiple rows as a group). A select cursor is declared for sequential only (regular cursor) or non-sequential (scroll cursor) retrieval of row information. In addition, you can declare a select cursor for update (initiating locking control for updated and deleted rows) or withhold (completing a transaction will not close the cursor). In ESQL/C and ESQL/COBOL, a cursor can be dynamic, meaning that it can be an identifier or a character/string variable. The term <i>cursor</i> also refers to the position indicator on a video terminal.
CURSOR STABILITY	A level of process isolation available through IBM Informix OnLine in which the database server must secure a shared lock on a fetched row before the row can be viewed. The database server retains the lock until it receives a request to fetch a new row. See <i>process isolation</i> .
current row	The most recently retrieved row of the active set of a query.

data integrity	The process of ensuring that data corruption does not occur when multiple users simultaneously try to alter the same data. Locking and transaction processing are used to control data integrity.
data type	A descriptor assigned to each column in a table or program variable indicating the type of data the column or program variable is intended to hold. Informix data types include SMALLINT, INTEGER, SERIAL, SMALLFLOAT, FLOAT, DECIMAL, MONEY, DATE, DATETIME, INTERVAL, CHAR, VARCHAR, TEXT, and BYTE.
database	A collection of information (contained in tables) that is useful to a particular organization or used for a specific purpose.
Database Administrator (DBA)	The individuals responsible for the contents and use of a database.
database application	A program that applies database management techniques to implement specific data manipulation and reporting tasks.
database dictionary	The collection of tables used by database management programs to keep track of the structure of the database. Information about the database is maintained in the database dictionary, which is sometimes referred to as the "data dictionary" or "system catalog."
database server process	The portion of the database management system that actually manipulates the database files. This is the process that receives SQL statements from the database application, parses them, optimizes the approach to the data, retrieves the data from the database, and returns the data to the application. The database server is sometimes referred to as the "back end" or "database engine."
database management system (DBMS)	All the components necessary to create and maintain a database, including the application development tools and the database server.
DBA-privileged	Refers to having the privileges generally associated with a DBA, even if not used by someone who actually has DBA privileges
DB-Monitor	An interface that presents a series of screens through which a System Administrator can monitor and modify an IBM Informix OnLine database server.
DBA	Acronym for Database Administrator.

dbspace	A logical collection of chunks that represents a region of disk space in IBM Informix OnLine. In some ways, the dbspace corresponds to a directory in the UNIX file system. For example, you can create a table in a particular dbspace.
deadlock	A situation in which two or more processes cannot proceed because each is waiting for a lock held by the other (or another) process. IBM Informix OnLine monitors and prevents potential deadlock situations by sending an error message to the process whose request for a lock would result in a deadlock. In distributed queries across multiple systems, IBM Informix STAR controls deadlocks. See <i>multisite deadlock</i> .
debug file	A file that receives output used for debugging purposes.
debugger	A software product to analyze programs and to detect and locate errors in program logic. The IBM Informix 4GL Interactive Debugger is a 4GL source language debugger that supports a wide variety of programming tools, such as tracing program logic and stopping execution at preset points. See <i>breakpoint</i> and <i>tracepoint</i> .
declarative statement	The programming language statements that describe or define objects, for example, defining a program variable. See <i>procedural statement</i> .
default	How a program acts if the user does not explicitly specify an action.
default value	A value inserted into a column when an explicit value is not specified. Default values can be assigned to columns using the ALTER TABLE and CREATE TABLE statements.
delimiter	A boundary on an input field or the terminator for a column or row. In a form specification, the field delimiters are square brackets ([]) and determine the size of the field. Some files and prepared objects require semicolon (;) delimiters between statements.
descriptor	A quoted string or embedded variable name that identifies an allocated system descriptor area or an sqllda structure. It is used for dynamic SQL statement management by IBM Informix 4GL and the IBM Informix embedded-language products. See <i>identifier</i> .

DIRTY READ	A level of process isolation that does not account for locks and does allow viewing of any existing rows, even rows that currently can be altered from inside an uncommitted transaction. DIRTY READ is the lowest level of isolation (no isolation at all). It is the level at which IBM Informix SE normally operates and it is an optional level under IBM Informix OnLine. See <i>process isolation</i> .
disk configuration	The spatial organization of a disk.
disk I/O	The process of transferring data between memory and disk.
display field	Used in a screen form to indicate where data is to be displayed on the screen. A display field usually is associated with a column in a table.
display label	A temporary name for a column or expression in a query.
distributed option	The ability to access data in multiple databases. The databases can be on the same hardware or on a computer network. (IBM Informix OnLine can perform multiple-database queries. IBM Informix STAR can query databases on more than one OnLine database server.)
dominant table	See <i>outer join</i> .
DOS engine	Portion of IBM Informix database products for the DOS environment that handle the execution of SQL statements. See also <i>database server</i> .
dynamic statements	SQL statements that are created at the time the program is executed, rather than when the program is written. The PREPARE statement in IBM Informix 4GL and the IBM Informix embedded-language products is used to create dynamic statements.
embedded SQL	SQL statements placed within a host language. Informix supports embedded SQL in C and COBOL.
environment variable	A variable with an assigned value that is maintained by the operating system and made available to all programs.
error message	A message displayed on the screen or written to a file to describe either the failure of an action or an illegal specification. Each error is identified by a (usually negative) designated number.
error log	A file that receives error information whenever a program runs.
error trapping	Code within a program that anticipates and reacts to run-time errors.

escape key	The key (usually marked ESC) used to terminate one mode and start another in most UNIX and DOS systems. On many terminals the key is the default Accept key (used to indicate when you are finished entering text in a query, add, update, or delete action) for PERFORM and IBM Informix 4GL screen forms.
exception	Refers to an error returned by the database server or a state initiated by a stored procedure statement.
exclusive access	The user has sole access to the information. Other users are prevented from using the database or table.
exclusive lock	A lock on an object (row, page, table, or database) held by a single process that prevents other processes from acquiring a lock of any kind on the same object.
executable file	A binary file containing compiled code that can be run as a program. May also refer to a UNIX shell script or a DOS batch file.
execute	To carry out a program or a set of instructions.
expression	Anything from a simple numeric or alphabetic constant to a more complex series of column values, functions, quoted strings, operators, and keywords. A Boolean expression contains a logical operator (>, <, =, !=, IS NULL, and so on) and evaluates as TRUE, FALSE, or UNKNOWN. An arithmetic expression contains the operators (+, -, ×, /, and so on) and evaluates as a number.
extent	A continuous segment of disk space allocated to a tblspace in IBM Informix OnLine. The programmer can specify both the initial extent size for a table and the size of all subsequent extents allocated to the table.
external table	A database table that is not in the current database.
fast recovery	An automatic, fault-tolerant feature that IBM Informix OnLine implements any time the operating mode changes from off-line to quiescent mode. The aim of fast recovery is to return OnLine to a state of physical and logical consistency with minimal loss of work in the event of a system failure.
fault tolerance	See <i>high availability</i> .
file	A collection of related information stored together on the system, such as the words in a letter or report, a computer program, or a listing of data.
filename extension	The part of a filename following the period. For example, DB-Access appends the extension .sql to command files.

file server	Network node that manages a set of disks and provides storage services to computers on the network.
fixchar	A character data type, available in IBM Informix ESQL/C programs, in which the character string is fixed in length, padded with blanks if necessary, and not null-terminated.
flag	A command-line option, usually indicated by a minus (-) sign in UNIX systems. For example, in DB-Access the -e flag echoes input to the screen.
floating-point number	A number with fixed precision (total number of digits) and undefined scale (number of digits to the right of the decimal point). The decimal point “floats” as appropriate to represent an assigned value.
footer	See <i>page trailer</i> .
forced residency	An option that forces UNIX to keep IBM Informix OnLine shared-memory segments resident in memory, preventing UNIX from swapping out these segments to disk. (This option is not available in all UNIX systems.)
foreign key	A column, or set of columns, that references a unique or primary key in a table. For every entry in a foreign-key column, there must exist a matching entry in the unique or primary column, if all foreign-key columns contain non-null values.
form specification	A system file containing instructions describing how a form looks and performs. You must compile a form specification before you can use it.
function	See <i>program block</i> .
global variable	A variable whose value you can access from any module or function in a program. See <i>variable</i> and <i>scope of reference</i> .
header	See <i>page header</i> .
help message	On-line text displayed automatically or at the request of the user to assist the user in interactive programs. Such messages are stored in help files.
hierarchy	A tree-like data structure in which some groups of data are subordinate to others such that (1) only one group (called <i>root</i>) exists at the highest level and (2) each group except <i>root</i> is related to only one group on a higher level.
high availability	The ability of a system to resist failure and loss of data. High availability includes features such as fast recovery and mirroring. It is sometimes referred to as “fault tolerance.”

highlight	An inverse-video rectangular area that marks your place on the screen. A highlight often indicates the current option on a menu or the current character in an editing session. If a terminal cannot display highlighting, the current option often appears in angle brackets, while the current character is underlined.
home page	The page that contains the first byte of the data row. Even if a data row outgrows its original storage location, the home page does not change. The home page contains a forward pointer to the new location of the data row. See <i>remainder page</i> .
home server	The first database server that the current database accesses in a distributed query across a network of IBM Informix STAR systems.
host variable	A C or COBOL program variable that is referenced in a statement. A host variable is identified by the dollar sign (\$) or colon (:) that precedes it.
identifier	A sequence of letters, digits, and underscores (_) that represents the name of a database, table, column, screen form, program variable, cursor, function, index, window, menu, synonym, alias, view, prepared object, constraint, report, or procedure name.
incremental archiving	A three-level system of archiving in IBM Informix OnLine that allows you the option to archive only those parts of the data that have changed since the last archive.
index	A file containing pointers to rows of data. Indexes can speed ordering of rows and optimize the performance of database queries.
infocmp	A UNIX program that you can use to view or decompile files in a terminfo directory, or that you can use to compare compiled terminfo entries with entries in a termcap file.
initialize	The act of assigning a starting value.
input	Information received from an external source (for example, from the keyboard, a file, or another program) and processed by a program.
installation	Loading software from some magnetic medium (tape, cartridge, floppy disk) onto the computer and preparing it for use.
interactive	Programs that accept input from the user, process the input, and then produce output on the screen, in a file, or on a printer.
interpreter	A program that reads, decodes, and executes statements one at a time.

interrupt	A signal from a user or another process that can stop the current process temporarily or permanently. See <i>signal</i> .
ISAM	Acronym for Indexed Sequential Access Method. An access method is a way of retrieving pieces of information (rows) from a larger set of information (table). An indexed sequential access method allows you to find information in a specific order or to find specific pieces of information quickly through an index.
join	The process of combining information from two or more tables based on some common domain of information. Rows from one table are paired with rows from another table when information in the corresponding rows match on the joining criterion. For example, if a customer_number column exists in both customer and orders tables, you can construct a query that pairs each customer row with all the associated orders rows based on the common customer_number . See <i>Cartesian product</i> and <i>outer join</i> .
kernel	The part of the UNIX operating system that controls processes and the allocation of resources.
key	The pieces of information used to locate a row of data. A key defines the pieces of information you want to search for, as well as the order in which you want to process information in a table. For example, you can index the last_name column in a customer table to find specific customers or to process the customers in alphabetical or reverse alphabetical order, according to last name.
keyword	A word that has meaning to a program. For example, the word SELECT is a keyword in SQL relating to database queries.
latch	Used by IBM Informix OnLine to coordinate user processes as they attempt to modify entries in shared memory.
level of isolation	See <i>process isolation</i> .
library	A collection of precompiled functions or routines designed to perform tasks common to a particular kind of application. Your product can include library functions or routines that you can call from your programs.
link	The process of combining separately compiled program modules into an executable program.

literal	A character constant. In the format string of a PICTURE attribute, for example, any characters except A, #, and X are literals because they are displayed exactly as they appear in the format string.
local variable	A variable that has meaning only in the module in which it is defined. See <i>variable</i> and <i>scope of reference</i> .
lock mode	An option that sets whether a user who requests a lock on an already locked object is to (1) not wait for the lock and instead receive an error, (2) wait until the object is released to receive the lock, or (3) wait a certain amount of time before receiving an error (an option available only with IBM Informix OnLine). In IBM Informix OnLine, the lock mode also can refer to the standard unit of locking (either page or row) chosen by the programmer.
locking	The process of temporarily limiting access to an object (database, table, page, or row) to prevent conflicting interactions among concurrent processes. Locks can be in either exclusive mode, which restricts both read and write access to only one user, or share mode, which allows read-only access to other users. In addition, there are update locks that begin in share mode but are upgraded to exclusive mode when a row is actually changed.
locking granularity	The size of an object that is locked. The size may be a database, table, page, or row.
logical log	An allocation of disk space that contains records of all changes that were performed on a database during the period the log was active. The logical log is used to roll back transactions, recover from system failures, and restore databases from archives. This log is also referred to as a "transaction log."
login	The process of identifying oneself to a computer.
macro	A name given to a set of instructions that the computer executes whenever the name is referenced.
mantissa	The significant digits in a floating point number, usually expressed as a number between zero and one.
menu	A screen display that allows you to choose the commands that you want the computer to perform.
message log	The UNIX file that IBM Informix OnLine keeps to record significant events, such as checkpoints, filling of log files, recovery data, and errors.

mirroring	Storing the same data on two chunks simultaneously. If one chunk fails, the data is still usable on the other chunk in the mirrored pair. This option is available with IBM Informix OnLine.
module	The part of a program that resides in a single system file. Each file represents one module. Entire programs can reside in a single module, or they can be contained in several modules, each performing a specific function or purpose.
module variable	A variable whose value you can reference from any program block within the same module but not from other modules. See <i>variable</i> and <i>scope of reference</i> .
monochrome	A term that describes a monitor that can display only one color.
multisite deadlock	A deadlock that occurs between processes due to a distributed query across multiple IBM Informix STAR systems.
null value	A value representing “unknown” or “not applicable.” (A null is not the same as a value of zero or blank.)
offset	A term used in IBM Informix OnLine to specify the physical position of a chunk on a disk. The offset is the number of kilobytes indented into the named device (which is the specified disk partition) before starting the chunk. The maximum allowed value of an offset is 2 Terabytes.
open	The process of making a resource available, such as preparing a file for access, activating a cursor, or initiating a window.
operating mode	Refers to IBM Informix OnLine states of operation. The five operating modes include off-line, quiescent, on-line, shutdown, and recovery.
outer join	An asymmetric joining of a dominant and a subservient table in a query, whereby joining restrictions apply only to the subservient or “outer” table. Rows in the dominant table are retrieved without considering the join, but rows from the outer table are included only if they also satisfy the join conditions. Any dominant-table rows that do not have a matching row from the outer table receive a row of nulls in place of an outer-table row.
output	The result that the computer produces in response to such things as a query or a request for a report.
owner-privileged	Refers to having the privileges held by the owner of the object, rather than those associated with the user.

pad	To fill empty places at the beginning or end of a line, string, or field, usually with a space or a blank.
page	The basic unit of disk and memory storage used by IBM Informix OnLine. The size is fixed for a port, and the customer cannot tune it.
page header	The items that are printed at the top of each page of a report (for example, the title and date). A “running header” appears at the top of each page of a report.
page trailer	The items that are printed at the bottom of each page of a report (for example, the page number). A page trailer is also referred to as a “footer.” A “running footer” appears at the bottom of each page of a report.
parameter	A variable that is given a constant value for a specified application. In a subroutine, a parameter commonly uses an argument value passed to that routine.
pattern	An identifiable or repeatable series of characters or symbols.
permission	On some operating systems, refers to the right to have access to files and directories.
phantom row	A row of a table that initially is modified or inserted during a transaction but subsequently is rolled back. Another process can see a phantom row if the isolation level is DIRTY READ. No other isolation level allows a changed but uncommitted row to be seen.
physical log	An allocation of disk space in IBM Informix OnLine that contains the before-images of all pages changed since the last checkpoint.
pointer	A number that specifies the address in memory of the data or variable of interest.
pop	Removes a value from a stack in memory. See <i>stack</i> and <i>push</i> .
precision	The total number of significant digits in a real number, both to the right and left of the decimal point. For example, the number 1437.2305 has a precision of 8. See <i>scale</i> .
prepared statement	An SQL statement that translates a character string created at run time into a request of the database. This feature allows you to form your request while the program is executing without having to modify and recompile the program.

preprocessor	A program that takes high-level programs and produces code that a standard language compiler, such as C or Micro Focus COBOL/2, can compile.
primary key	The information from a column or set of columns that uniquely identifies each row in a table. The primary key sometimes is called a “unique key.”
primary key constraint	Specifies that each entry in a column or set of columns contains a non-null unique value.
printable character	A character that can be displayed on a terminal or printer. Includes A-Z, a-z, 0-9, and punctuation. See <i>control character</i> .
privilege	The right to use or change the contents of a database.
procedural statements	The programming language statements that specify actions, for example, looping and branching if a condition is met. See <i>declarative statement</i> .
procedure	See <i>program block</i> .
process	See <i>database server process</i> .
process isolation	The level of process independence among multiple users when they attempt to access common data, specifically relating to the locking strategy for read-only SQL requests. The various levels of isolation are distinguished primarily by the length of time that shared locks are (or can be) acquired and held. IBM Informix SE sets a level of <i>no isolation</i> (referred to as a DIRTY READ), which cannot be changed. IBM Informix OnLine allows the programmer to choose from four levels of isolation. See <i>DIRTY READ</i> , <i>COMMITTED READ</i> , <i>CURSOR STABILITY</i> , and <i>REPEATABLE READ</i> .
program block	A named collection of statements that performs a particular task; a unit of program code. In 4GL, it refers to a MAIN, FUNCTION, REPORT, or GLOBALS section. A program block is sometimes referred to as a “function,” “procedure,” “division,” or “routine” (although these terms in some languages have subtle but distinct differences in meaning).
program control	Actions that the computer takes, as opposed to actions that the user takes.
projection	Taking a vertical subset from the columns of a single table that retains the unique rows. Projection is implemented through the select list in the SELECT clause of a SELECT statement and returns some of the columns and all of the rows in a table. See <i>selection</i> and <i>join</i> .
promotable lock	A lock that can be changed from a shared lock to an exclusive lock. See <i>update lock</i> .

protocol	A set of rules that govern communication between computers. These rules govern format, timing, sequencing, and error control.
push	Refers to placing a value onto a stack in memory. See <i>stack</i> and <i>pop</i> .
query	A request to the database to retrieve data that meets certain criteria.
raw device	A UNIX disk partition that is defined as a character-special device and that is not mounted.
record	See <i>row</i> .
recover a database	To restore a database to a former condition after a system failure or other destructive event. The recovery restores the database as it existed immediately before the crash. This is sometimes referred to as “a data restore.”
referential constraint	Defines the relationship between columns within a table or between tables. It has a one-to-many relationship between the referenced columns and referencing columns. The referenced column must be part of a primary key or a unique constraint. The referencing columns can contain null values, but every non-null value in the referencing column must match a value in the referenced column if all columns are non-null. Referencing columns also are known as foreign keys.
relation	See <i>table</i> .
relational database	A database that uses a table structure to store data. Relationships among tables are specified logically at the time of user access into the database; they are not built into the data structures themselves (unlike some other database systems).
remainder page	An additional page filled with data from a single row. IBM Informix OnLine uses remainder pages when the data for a row cannot fit in the initial page. Remainder pages are added and filled as needed. The original page entry contains pointers to the remainder pages. See <i>home page</i> .
REPEATABLE READ	A level of process isolation available through IBM Informix OnLine that ensures all data read during a transaction is not modified by another process. Transactions under REPEATABLE READ also are known as <i>serializable</i> transactions. REPEATABLE READ is the default level of isolation under IBM Informix OnLine for ANSI-compliant databases. See <i>process isolation</i> .
report specification	A file or program segment that contains the description of a report. The report is described in a report-writing language.

report writer	A program, such as ACE, that allows the user to describe the appearance of a report using a report-writing language. The report writer then can compile this report specification into an executable report.
reserved word	A word in a statement or command that you cannot use in any other context of the language or program without receiving a warning or error message.
restore a database	See <i>recover a database</i> .
roll back	The process that reverses an action or series of actions upon a database. The database is returned to the condition that existed before the statements were executed. See <i>transaction</i> .
roll forward	The process that brings a database up to date. This process usually takes place when a database is <i>recovered</i> after a system crash or other failure. In IBM Informix SE, an archive copy of the database is restored to the disk, and the database is rolled forward to a point just before the failure. In IBM Informix OnLine, an archive copy of the database is restored to the disk, and the logical log records are rolled forward to a point just before the failure.
root dbspace	The initial dbspace for an IBM Informix OnLine system. In addition to any data, the root dbspace contains all system management tables.
routine	See <i>program block</i> .
row	A group of related items of information about a single entity in a database table. In a table of customer information, for example, a row contains information about a single customer. A row sometimes is referred to as a "record" or "tuple." (In a screen form, <i>row</i> can refer to a line of the screen.)
run-time errors	Refers to errors that occur during program execution. See <i>compile-time errors</i> .
run-time environment	The hardware and operating system services available at the time a program runs.
scale	The number of digits to the right of the decimal place in DECIMAL notation. The number 1437.2350 has a scale of 4 (four digits to the right of the decimal point). See <i>precision</i> .
schema	A listing of the structure of a database or a table. The schema for a table lists the names of the columns, their data types and (where applicable) lengths, indexing, and other information about the structure of the table.

scope of reference	The portion of a program in which an identifier applies and can be accessed. There are three possible scope sizes: local (an identifier applies only within a single program block), modular (the identifier applies throughout a single module), and global (an identifier applies throughout the entire program). Identifiers must be declared before you can reference them. For example, a module identifier cannot be referenced prior to the statement within the module that declares it.
screen form	A data-entry form that is displayed on the screen of a terminal. The user enters data into the blanks on the form.
selection	Refers to taking the horizontal subset of rows of a single table that satisfies a particular condition. Selection is implemented through the WHERE clause of a SELECT statement and returns some of the rows and all of the columns in a table. See <i>projection</i> and <i>join</i> .
self-join	A join between a table and itself. A self-join occurs when a table is used two or more times in a SELECT statement (with different aliases) and joined to itself.
semaphore	A UNIX communication device that signals a user process to wake.
serializable transactions	See <i>REPEATABLE READ</i> .
server name	The unique name that the Database Administrator assigns to an IBM Informix OnLine system at the time of initialization. The database server name is used to identify external tables and databases.
server number	A unique number between 0 and 255 that the Database Administrator assigns to an IBM Informix OnLine database server at the time of initialization. If more than one IBM Informix OnLine database server is installed on the same machine, each database server must have a unique number.
shared lock	A lock that more than one process can acquire on the same object. Shared locks allow for greater concurrency with multiple users; if two users have locks on a row, a third user cannot change the contents of that row until both users (not just the first) release the lock. Shared-locking strategies are used in all levels of process isolation except DIRTY READ.
shared memory	Allows multiple processes to access a common data space in memory. Common data does not have to be reread from disk for each process, reducing disk I/O and improving performance.

signal	A special character or set of characters used as a means of communication between two processes. For example, signals are sent when a user or a program wishes to interrupt or suspend the execution of a process. See <i>interrupt</i> .
singleton select	A SELECT statement that returns a single row.
source file	A file containing instructions (in ASCII text) that is used as the source for generating compiled programs.
SPL	An acronym for Stored Procedure Language.
SQL	Acronym for Structured Query Language. A database query language developed by IBM and standardized by ANSI. IBM Informix relational database management products are based on an extended implementation of ANSI-standard SQL.
SQLCA	Acronym for SQL Communications Area. It is a data structure that stores information about the most recently executed SQL statement. The result code returned by the database server to the SQLCA is used for error handling by IBM Informix 4GL and the IBM Informix embedded-language products.
sqllda	Acronym for SQL Descriptor Area. It is a structure that contains an array of data descriptors that hold descriptive information about values used by dynamic SQL statements. It can be used by ESQL/C. See <i>descriptor</i> .
sqlxecd	The IBM Informix NET process that receives requests from clients and spawns a database server process to access the data.
stack	An area of memory reserved for the temporary storage of data elements used by a program. (There can be more than one stack.) The stack usually holds data that is of immediate use to the program, such as the arguments passed to a function. Items are removed from the stack in the reverse order from which they were inserted. See <i>push</i> and <i>pop</i> .
stack operator	Operators that allow programs to manipulate values that are on the stack.
statement	A line, or set of lines, of program code that describes a single action (for example, a SELECT statement or an UPDATE statement).
statement block	Refers to a section of a program, usually beginning and terminating with special symbols such as "begin" and "end." A statement block is the smallest unit of scope of reference for program variables.

statement identifier	An embedded variable name or SQL statement identifier that represents a data structure defined in a PREPARE statement. It is used for dynamic SQL statement management by IBM Informix 4GL and the IBM Informix embedded language products.
status variable	A program variable that indicates the status of some aspect of program execution. Status variables often store error numbers or act as flags to indicate that an error has occurred.
stored procedure	Refers to a function used along with SQL statements in an Informix program. Stored procedures are written using SQL and SPL statements. The procedure is stored in the database in executable form.
Stored Procedure Language	A language developed by Informix for use in stored procedures.
string	A set of characters (generally alphanumeric) that is manipulated as a single unit. A string might consist of a word (such as “Smith”), a set of digits representing a number (such as “19543”), or any other collection of characters. Strings generally are surrounded by single or double quotes. Also a character data type, available in IBM Informix ESQL/C programs, in which the character string is stripped of trailing blanks and is null-terminated.
subquery	A query that is embedded as part of another SQL statement. For example, an INSERT statement can contain a subquery in which a SELECT statement supplies the inserted values in place of a VALUES clause; an UPDATE statement can contain a subquery in which a SELECT statement supplies the updating values; or a SELECT statement can contain a subquery in which a second SELECT statement supplies the qualifying conditions of a WHERE clause for the first SELECT statement. (Parentheses always delimit a subquery, unless you are referring to a CREATE VIEW statement or UNIONS.)
subservient table	See outer join.
synonym	A name assigned to a table and used in place of the original name for that table. A synonym does not replace the original table name; instead, it acts as an alias for the table.
system call	A call to a function provided by the operating system.
system catalog	Database tables that contain information about the database itself, such as the names of tables or columns in the database, the number of rows in a table, information about indexes and database privileges, and so forth.

system descriptor area	A descriptor used in SQL that contains descriptive information about database columns or host variables used in dynamic SQL statements. A system descriptor area can be used in ESQL/C, and ESQL/COBOL.
table	A rectangular array of data in which each row describes a single entity and each column contains the values for each category of description. For example, a table can contain the names and addresses of customers. Each row corresponds to a different customer and the columns correspond to the name and address items. A table sometimes is referred to as a relation.
tblspace	The logical collection of extents assigned to a table in IBM Informix OnLine.
TCP/IP	Transmission Control Protocol/Internet Protocol. A popular network protocol used in DOS, UNIX, and other environments.
temporary	An attribute of any file, index, or table that is deleted when program execution terminates or an on-line session ends.
termcap	An ASCII file that contains the names and capabilities of common terminals.
terminfo	A hierarchical directory structure that contains compiled files of terminal capabilities.
tic	A UNIX program that compiles terminfo source files or terminfo files that have been decompiled using infocmp. See <i>infocmp</i> .
timeout	The point at which a lock request is aborted because the requesting process waited longer for the lock than the specified maximum time limit. A program developer can set a time limit in IBM Informix OnLine through the SET LOCK MODE statement; the Database Administrator sets a time limit for operations across multiple IBM Informix STAR systems.
TLI	Transport Level Interface. Interface designed for use by application programs that are independent of network protocol.
trace	Refers to keeping a running list of the values of program variables, arguments, expressions, and so on, in a stored procedure.
tracepoint	A named object, specified by a debugger, that the programmer can associate with a statement, program block, or variable. When the tracepoint is reached, the debugger displays information about the associated statement, program block, or variable, and executes any optional commands that are specified by the programmer. A tracepoint must be enabled to take effect. See <i>debugger</i> .

transaction	A collection of one or more SQL statements that is treated as a single unit of work. If one of the statements in a transaction fails, the entire transaction can be rolled back (canceled). If the transaction is successful, the work is committed and all changes to the database from the transaction are accepted.
transaction log	See <i>logical log</i> .
tuple	See <i>row</i> .
unique constraint	Specifies that each entry in a column or set of columns contains a unique value.
unique key	See <i>primary key</i> .
unlock	To free an object (database, table, page, or row) that has been locked. For example, a locked table prevents others from adding, removing, updating, or (in the case of an exclusive lock) viewing rows in that table, as long as it is locked. When the user or program unlocks the table, others are again permitted access.
update lock	A promotable lock acquired during a <code>SELECT...FOR UPDATE</code> . An update lock behaves like a shared lock until the update actually occurs; it then becomes an exclusive lock. It differs from a shared lock in that only one update lock can be acquired on an object at a time.
user interface	The part of the program that communicates with the user by prompting for input and displaying output based on information the user enters. Typical user interfaces include menus, prompts, screen forms, and on-line help messages.
variable	The identifier for a location in memory that stores the value of a program object whose value can change during the execution of the program.
view	A dynamically controlled picture of the contents in a database that allows you to determine what information the user sees and manipulates. A view represents a virtual table based on a specified <code>SELECT</code> statement.
virtual column	A derived column of information that is not stored in the database. For example, you can create virtual columns in a <code>SELECT</code> statement by arithmetically manipulating a single column, such as multiplying existing values by a constant, or by combining multiple columns, such as adding the values from two columns.

- warning** A state or situation detected by the database server or compiler, possibly incorrect syntax or a problem. A warning does not necessarily affect the ability of the code to run.
- wildcard** A special symbol that represents any sequence of zero or more characters or any single character. In SQL, for example, the asterisk (*), question mark (?), brackets ([]), percent sign (%), and underscore (_) can be used as wildcard characters. (The asterisk, question mark, and brackets are also wildcards in UNIX.)
- window** A rectangular area on the screen in which you can take actions without leaving the context of the background program.

Index

A

- ABSOLUTE keyword
 - syntax in FETCH 7-153
 - use in FETCH 7-156
- ACCESS FOR keywords, in INFO statement 7-187
- Active set
 - constructing with OPEN 7-208
 - retrieving data with
 - FETCH 7-153
- ADD CONSTRAINT keywords, syntax in ALTER TABLE 7-20
- Aggregate function
 - in 4GL and ESQL 7-397
 - in EXISTS subquery 7-356
 - in expressions 7-262, 7-383
 - in SELECT 7-263
 - in SPL expressions 8-23
 - restrictions with GROUP BY 7-280
 - summary 7-396
- Alias
 - for a table in SELECT 7-270
- ALL keyword
 - beginning a subquery 7-276
 - syntax
 - in expression 7-383
 - in GRANT 7-179
 - in REVOKE 7-249
 - in SELECT 7-260
 - with UNION operator 7-258
 - use
 - in Condition subquery 7-357
 - in expression 7-394
 - in GRANT 7-180
 - in REVOKE 7-250
- in SELECT 7-261
 - with UNION operator 7-288
- ALLOCATE DESCRIPTOR statement
 - syntax 7-13
 - with concatenation operator 7-372
- Allocating memory
 - dynamically in ESQL/C 6-4
 - dynamically in ESQL/COBOL 6-13
 - with the ALLOCATE DESCRIPTOR statement 7-13
- ALTER INDEX statement
 - creating clustered index 7-17
 - dropping clustered index 7-19
 - syntax 7-17
- ALTER keyword
 - syntax
 - in GRANT 7-175
 - in REVOKE 7-249
 - use
 - in GRANT 7-180
 - in REVOKE 7-250
- Alter privilege 7-180
- ALTER TABLE statement
 - ADD clause 7-22
 - ADD CONSTRAINT clause 7-33
 - adding a column 7-22
 - adding a column constraint 7-33
 - changing column data type 7-30
 - changing table lock mode 7-36
 - CHECK clause 7-28
 - DEFAULT clause 7-23
 - DROP clause 7-29
 - DROP CONSTRAINT clause 7-35
 - dropping a column 7-29

dropping a column
 constraint 7-35
 LOCK MODE clause 7-36
 MODIFY NEXT SIZE clause 7-36
 modifying next size 7-36
 PAGE keyword 7-36
 privilege for 7-175
 recluster a table 7-18
 REFERENCES clause 7-27
 ROW keyword 7-36
 rules for primary key
 constraints 7-34
 rules for unique constraints 7-34
 AND keyword
 syntax in Condition
 segment 7-345
 use
 in Condition segment 7-358
 with BETWEEN keyword 7-272
 AND logical operator 7-358
 ANSI compliance
 -ansi flag 4-7, 7-69, 7-76, 7-98
 level Intro-22
 table naming 7-243
 updating rows 7-326
 ANSI-compliant database
 create with START
 DATABASE 7-317
 description of 7-51
 FOR UPDATE not required
 in 7-111
 index naming 7-360, 7-413, 7-433
 procedure naming 7-424
 table privileges 7-76
 using
 with IBM Informix SE 7-53
 with BEGIN WORK 7-38
 ANY keyword
 beginning a subquery 7-276
 in WHENEVER 7-337, 7-339
 use in Condition subquery 7-357
 Arithmetic operator, in
 expression 7-371
 Array, moving rows into with
 FETCH 7-158
 AS keyword
 in SELECT 7-260
 syntax
 in CREATE VIEW 7-97

in GRANT 7-175
 use
 in CREATE VIEW 7-99
 in GRANT 7-182
 with display labels 7-264
 with table aliases 7-271
 ASC keyword
 syntax
 in CREATE INDEX 7-54
 in SELECT 7-283
 use
 in CREATE INDEX 7-56
 in SELECT 7-284
 ASCII collating order 7-430
 Asterisk (*)
 arithmetic operator 7-371
 use in SELECT 7-260
 At (@) sign, in database name 7-362
 Audit trail
 applying with RECOVER
 TABLE 7-238
 dropping with DROP
 AUDIT 7-131
 manipulating audit trail file 7-239
 no clustered index 7-56
 starting with CREATE
 AUDIT 7-47
 AVG function
 syntax in expression 7-383
 use in expression 7-395

B

Backslash (\)
 as escape character with
 LIKE 7-353
 as escape character with
 MATCHES 7-353
 BEGIN WORK statement
 locking in a transaction 7-37
 syntax 7-37
 BETWEEN keyword
 syntax in Condition
 segment 7-346
 use
 in Condition segment 7-349
 in SELECT 7-272

Binary Large Object (BLOB)
 effect of isolation on
 retrieval 7-309
 in a LOAD statement 7-202
 in an UNLOAD statement 7-321
 Boldface type Intro-11
 Boolean expression
 in Condition segment 7-345
 Bourne shell, how to set
 environment variables 4-4
 BUFFERED keyword, syntax in SET
 LOG 7-313
 BUFFERED LOG keywords
 syntax in CREATE
 DATABASE 7-49
 use in CREATE DATABASE 7-52
 Buffered logging 7-49
 BYTE data type
 considerations for UNLOAD
 statement 7-321
 description of 3-5
 requirements for LOAD
 statement 7-202
 selecting a BYTE column 3-6
 syntax 7-365
 with stored procedures 8-44, 8-48

C

C shell, how to set environment
 variables 4-4
 Calculated expression
 description of 7-383
 restrictions with GROUP
 BY 7-280
 CALL keyword, in the
 WHENEVER statement 7-337,
 7-342
 CALL statement
 assigning values with 8-25
 executing a procedure 8-12
 syntax 8-37
 call_type table in stores5 database,
 columns in 1-10
 Caret (^) wildcard 7-353
 CHAR data type
 changing data types 3-22
 description of 3-6

- in INSERT 7-428
 - syntax 7-365
 - using as default value 7-24, 7-82
- Character string
 - as DATE values 3-29
 - as DATETIME values 3-11, 3-29
 - as INTERVAL values 3-16
- Check constraint
 - adding with ALTER TABLE 7-28
 - creating with CREATE TABLE 7-89
 - definition of 7-89
 - specifying at column level 7-89
 - specifying at table level 7-89
- CHECK keyword
 - use in ALTER TABLE 7-28
 - use in CREATE TABLE 7-89
- CHECK TABLE statement, syntax and usage 7-39
- Checking for corrupted tables 7-39
- CLOSE DATABASE statement
 - prerequisites to close 7-44
 - syntax 7-44
- CLOSE statement
 - closing a select cursor 7-42
 - closing an insert cursor 7-42
 - cursor affected by transaction end 7-43
 - syntax 7-41
 - with concatenation operator 7-372
- CLUSTER keyword
 - syntax
 - in ALTER INDEX 7-17
 - in CREATE INDEX 7-54
 - use
 - in ALTER INDEX 7-18
 - in CREATE INDEX 7-55
- Clustered index
 - creating with CREATE INDEX 7-55
 - with ALTER INDEX 7-17
 - with audit trails 7-56
- Colon (:)
 - as delimiter in DATETIME 3-10
 - as delimiter in INTERVAL 3-15
- Color, setting INFORMIXTERM for 4-23
- Column
 - changing data type 3-22
 - defining as foreign key 7-85
 - defining as primary key 7-85
 - displaying information for 7-186
 - dropping with ALTER TABLE 7-29
 - in stores5 database 1-5 to 1-11
 - inserting into 7-191
 - modifying with ALTER TABLE 7-30
 - naming conventions 7-80, 7-241
 - number allowed when defining constraint 7-77
 - putting a constraint on 7-77
 - referenced and referencing 7-27, 7-86
 - renaming 7-241
 - specifying check constraint for 7-89
 - specifying with CREATE TABLE 7-80
 - virtual 7-99
- Column expression
 - in SELECT 7-262
 - syntax 7-373
- Column-level privilege 7-180
- COLUMNS FOR keywords, in INFO statement 7-186
- COMMIT WORK statement
 - syntax 7-46
- Committed Read isolation
 - level 7-308
- COMMITTED READ keywords, syntax in SET ISOLATION 7-307
- Comparison condition
 - syntax and use 7-346
- Compiler
 - setting environment variable for COBOL 4-20, 4-21
 - specifying storage mode for COBOL 4-21
- Compliance
 - with industry standards Intro-22
- Composite column list, multiple-column restrictions 7-34
- Composite index
 - column limit 7-56
 - creating with CREATE INDEX 7-54
 - definition of 7-56
- Compound assignment 8-64
- Concatenation operator (||) 7-372
- Concurrency
 - Committed Read isolation 7-308
 - Cursor Stability isolation 7-309
 - defining with SET ISOLATION 7-307
 - Dirty Read isolation 7-308
 - Repeatable Read isolation 7-309
- Condition segment
 - ALL, ANY, SOME subquery 7-357
 - boolean expressions 7-346
 - comparison condition 7-346
 - description of 7-345
 - join conditions 7-277
 - relational operators in 7-348
 - subquery in SELECT 7-355
 - syntax 7-345
 - use of functions in 7-346
 - wildcards in searches 7-353
 - with BETWEEN keyword 7-349
 - with ESCAPE keyword 7-354
 - with EXISTS keyword 7-356
 - with IN keyword 7-350
 - with IS keyword 7-351
 - with LIKE keyword 7-352
 - with MATCHES keyword 7-352
 - with NOT keyword 7-352
- CONNECT keyword
 - in GRANT 7-177
 - in REVOKE 7-251
- Connect privilege 7-177, 7-251
- Constant expression
 - in SELECT 7-262
 - inserting with PUT 7-232
 - restrictions with GROUP BY 7-280
 - syntax 7-376
- Constraint
 - adding with ALTER TABLE 7-33, 7-77
 - definition of 7-76
 - dropping with ALTER TABLE 7-29, 7-35, 7-77
 - enforcing 7-78

- modifying a column that has constraints 7-30
- number of columns allowed 7-77, 7-84
- privileges needed to create 7-35
- requirements for 7-25
- rules for unique constraints 7-34
- setting checking mode 7-289
- specifying at table level 7-84
- specifying at the column level 7-83
 - with DROP INDEX 7-134
- CONSTRAINT keyword
 - in ALTER TABLE 7-33
 - in CREATE TABLE 7-83
- Contact information Intro-23
- CONTINUE keyword, in the WHENEVER statement 7-337, 7-342
- CONTINUE statement
 - exiting a loop 8-27
 - syntax 8-40
- Conventions
 - example code Intro-17
 - for naming tables 7-76
 - syntax Intro-12
 - typographical Intro-11
- Converting data types 3-22
- Correlated subquery
 - definition of 7-355
- COUNT field
 - getting contents with GET DESCRIPTOR 7-169
 - setting value for WHERE clause 7-293
 - use in GET DESCRIPTOR 7-171
- COUNT function
 - use in expression 7-394, 7-396
- COUNT keyword, use in SET DESCRIPTOR 7-295
- CREATE AUDIT statement
 - need for archive 7-48
 - starts audit trail 7-47
 - syntax 7-47
- CREATE DATABASE statement
 - ANSI compliance 7-51
 - logging with OnLine 7-51
 - syntax 7-49
 - using with
 - CREATE SCHEMA 7-68
 - IBM Informix SE 7-52
 - PREPARE 7-50
- CREATE INDEX statement
 - composite indexes 7-56
 - implicit table locks 7-55
 - syntax 7-54
 - using
 - with ASC keyword 7-56
 - with CLUSTER keyword 7-55
 - with CREATE SCHEMA 7-68
 - with DESC keyword 7-56
 - with UNIQUE keyword 7-55
- CREATE PROCEDURE FROM statement
 - in embedded languages 8-8
 - syntax and usage 7-67
- CREATE PROCEDURE statement
 - inside a CREATE PROCEDURE FROM 8-8
 - syntax 7-58
 - using 8-7
- CREATE SCHEMA statement
 - syntax 7-68
 - with CREATE sequences 7-69
 - with GRANT 7-69
- CREATE SYNONYM statement
 - ANSI-compliant naming 7-71
 - chaining synonyms 7-73
 - privileges on synonym 7-70
 - synonym for a table 7-70
 - synonym for a view 7-70
 - syntax 7-70
 - with CREATE SCHEMA 7-68
- CREATE TABLE statement
 - CHECK clause 7-89
 - creating temporary table 7-90
 - DEFAULT clause 7-81
 - defining constraints
 - at column level 7-83
 - at table level 7-84
 - IN dbspace clause 7-92
 - LOCK MODE clause 7-95
 - naming conventions 7-76
 - REFERENCES clause 7-86
 - rules for primary keys 7-86
 - rules for referential constraints 7-86
 - rules for unique constraints 7-86
 - setting columns NOT NULL 7-25, 7-83
 - specifying extent size 7-94
 - specifying table columns 7-80
 - storing database tables 7-92
 - syntax 7-75
 - with BLOB data types 7-84
 - with CREATE SCHEMA 7-68
- CREATE VIEW statement
 - column data types 7-98
 - privileges 7-98
 - syntax 7-97
 - virtual column 7-99
 - WITH CHECK OPTION 7-100
 - with CREATE SCHEMA 7-68
 - with SELECT * notation 7-98
- Current database
 - specifying with DATABASE 7-101
- CURRENT function
 - syntax
 - in Condition segment 7-346
 - in expression 7-376
 - in INSERT 7-194
 - use
 - in ALTER TABLE 7-23
 - in CREATE TABLE 7-81
 - in expression 7-379
 - in INSERT 7-196
 - in WHERE condition 7-380
 - input for DAY function 7-380
- CURRENT keyword
 - syntax in FETCH 7-153
 - use in FETCH 7-156
- CURRENT OF keywords
 - syntax
 - in DELETE 7-122
 - in UPDATE 7-325
 - use
 - in DELETE 7-123
 - in UPDATE 7-333
- Cursor
 - activating with OPEN 7-207
 - affected by transaction end 7-43
 - associating with prepared statements 7-117
 - characteristics 7-111
 - closing 7-41

- closing with ROLLBACK WORK 7-255
- declaring 7-107
- definition of types 7-110
- opening 7-208
- retrieving values with FETCH 7-153
- scroll 7-112
- sequential 7-111
- use with transactions 7-118
- with
 - DELETE 7-122
 - INTO keyword in SELECT 7-266
 - prepared statements 7-110
- Cursor manipulation statements 7-10
- Cursor Stability isolation level 7-309
- CURSOR STABILITY keywords, syntax in SET ISOLATION 7-307
- Cursory procedure 8-57
- customer table in stores5 database, columns in 1-5
- cust_calls table in stores5 database, columns in 1-10

D

- Data access statements 7-11
- Data definition statements 7-10
- DATA field
 - description of 6-7, 6-16
 - setting with SET DESCRIPTOR 7-298
- Data integrity statements 7-11
- Data manipulation statements 7-10
- Data type
 - BYTE 3-5
 - changing with ALTER TABLE 7-32
 - CHAR 3-6
 - CHARACTER 3-7
 - considerations for INSERT 7-195, 7-428
 - conversion 3-22
 - DATE 3-7

- DATETIME 3-8
- DEC 3-11
- DECIMAL 3-11
- DOUBLE PRECISION 3-12
- FLOAT 3-12
- floating-point 3-12
- in SPL variables 8-20
- INT 3-13
- INTEGER 3-13
- INTERVAL 3-13
- MONEY 3-17
- NUMERIC 3-17
- REAL 3-17
- requirements for referential constraints 7-28, 7-88
- SERIAL 3-18
- SMALLFLOAT 3-19
- SMALLINT 3-19
- specifying with CREATE VIEW 7-98
- summary table 3-4
- syntax 7-365
- TEXT 3-19
- VARCHAR 3-21
- Data Type segment 7-365
- Database
 - closing with CLOSE DATABASE 7-44
 - creating ANSI-compliant 7-317
 - creating with CREATE DATABASE 7-49
 - data types 3-4
 - default isolation levels 7-307
 - dropping 7-132
 - map of
 - stores5 1-11
 - system catalog tables 2-33
 - naming conventions 7-363
 - naming with variable 7-363
 - opening in exclusive mode 7-103
 - optimizing queries 7-336
 - remote 7-363
 - restoring 7-256
 - stores5 Intro-6
- Database Administrator (DBA) 7-178
- Database lock 7-103
- Database Name segment
 - database outside DBPATH 7-364

- for remote database 7-363
- naming conventions 7-362
- naming with variable 7-363
- syntax 7-362
- using quotes, slashes 7-364
- DATABASE statement
 - determining database type 7-101
 - exclusive mode 7-103
 - for database outside
 - DBPATH 7-102
 - specifying current database 7-101
 - SQLAWARN after 7-102
 - syntax 7-101
 - with
 - LIKE in 4GL 7-103
 - program variables 7-102
- Database-level privilege
 - description of 7-177
 - granting 7-177
 - passing grant ability 7-181
 - revoking 7-251
- Database, stores5 description of 1-3
- Data, inserting with the LOAD statement 7-199
- DATE data type
 - converting to DATETIME 3-24
 - description of 3-7
 - range of operations 3-25
 - representing DATE values 3-28
 - syntax 7-365
 - with DATETIME and INTERVAL values 3-28
- Date data type
 - functions in 7-383
- DATE function
 - syntax in expression 7-383
 - use in expression 7-387
- DATE value
 - setting DBDATE environment variable 4-8
 - specifying European format with DBDATE 4-9
- DATETIME data type
 - adding or subtracting INTERVAL values 3-27
 - as quoted string 7-427
 - character string values 3-11
 - converting to DATE 3-24
 - field qualifiers 3-8, 7-368

- in
 - expression 7-381
 - INSERT 7-428
 - multiplying values 3-26
 - precision and size 3-8
 - range of expressions 3-25
 - range of operations with DATE and INTERVAL 3-25
 - representing DATETIME values 3-29
 - syntax 3-8, 7-365, 7-417
 - using the DBTIME environment variable 4-17
 - with EXTEND function 3-26, 3-27
- DATETIME Field Qualifier segment 7-368
- DATETIME formats, using the DBTIME environment variable 4-17
- DAY function
 - syntax in expression 7-383
 - use
 - in expression 7-385
- DAY keyword
 - syntax
 - in DATETIME data type 7-368
 - in INTERVAL data type 7-414
 - use
 - as DATETIME field qualifier 3-8, 7-417
 - as INTERVAL field qualifier 3-14, 7-420
- DBA keyword
 - in GRANT 7-178
 - in REVOKE 7-251
- DBANSIWARN environment variable 4-7, 7-69, 7-76, 7-98
- DBA-privileged procedure 8-16
- DBDATE environment variable 4-8
- DBDELIMITER environment variable 4-9, 7-203
- DBEDIT environment variable 4-10
- DBFORMAT environment variable 4-10
- DBLANG environment variable 4-11
- dbload utility
 - specifying field delimiter with DBDELIMITER 4-9
- DBMENU environment variable 4-12
- DBMONEY environment variable 4-12
- DBNETTYPE environment variable 4-13
- DBPATH environment variable 4-14, 7-102, 7-364
- DBPRINT environment variable 4-15
- DBREMOTECMD environment variable 4-15
- .dbs extension 7-50, 7-102
- DBSERVERNAME function
 - returning servername 7-378
 - use
 - in ALTER TABLE 7-23
 - in CREATE TABLE 7-81
 - in expression 7-378
- dbspace
 - selecting with CREATE DATABASE 7-49
- DBSRC environment variable 4-16
- DBTEMP environment variable 4-17
- DBTIME environment variable 4-17
- Deadlock detection 7-312
- DEALLOCATE DESCRIPTOR statement
 - syntax 7-105
 - with concatenation operator 7-372
- DECIMAL data type
 - changing data types 3-22
 - description of 3-11
 - floating-point 3-12
 - syntax 7-365
 - using as default value 7-24, 7-81
- Decimal point (.)
 - as delimiter in DATETIME 3-10
 - as delimiter in INTERVAL 3-15
- DECLARE statement
 - cursor characteristics 7-111
 - cursor types 7-110
 - cursors with prepared statements 7-117
 - cursors with transactions 7-118
- definition and use
 - hold cursor 7-112
 - insert cursor 7-111, 7-120
 - scroll cursor 7-112
 - select cursor 7-110
 - sequential cursor 7-111
 - update cursor 7-111, 7-114
- insert cursor with hold 7-120
- restrictions with SELECT with ORDER BY 7-285
- syntax 7-107
- updating specified columns 7-116
- with
 - concatenation operator 7-372
 - FOR UPDATE keywords 7-111
 - SELECT 7-267
- Default assumptions for your environment 4-3
- Default value
 - specifying with CREATE TABLE 7-81
 - specifying with ALTER TABLE 7-24
- Default values
 - specifying with ALTER TABLE 7-24
- Deferred checking 7-289
- DEFERRED keyword, in the SET CONSTRAINTS statement 7-289
- DEFINE statement
 - placement of 8-43
 - syntax 8-42
- DELETE keyword
 - syntax
 - in GRANT 7-179
 - in REVOKE 7-249
 - use
 - in GRANT 7-180
 - in REVOKE 7-250
- Delete privilege 7-179
- DELETE statement
 - CURRENT OF clause 7-123
 - privilege for 7-179
 - syntax 7-122
 - with Condition segment 7-345
 - with cursor 7-114
 - within a transaction 7-122

Delimiter
 for DATETIME values 3-10
 for INTERVAL values 3-15
 for LOAD input file 7-203
 specifying with UNLOAD 7-322

DELIMITER keyword
 in LOAD 7-203
 in UNLOAD 7-322

Demonstration database
 copying Intro-7
 installation script Intro-6
 map of 1-11
 overview Intro-6
 structure of tables 1-4
 tables in 1-5 to 1-11

DESC keyword 7-284
 syntax
 in CREATE INDEX 7-54
 in SELECT 7-283
 use
 in CREATE INDEX 7-56
 in SELECT 7-284

DESCRIBE statement
 describing statement type 7-126
 relation to GET
 DESCRIPTOR 7-172
 syntax 7-125
 the USING SQL DESCRIPTOR
 clause 7-128
 values returned by SELECT 7-127
 with concatenation
 operator 7-372

Dirty Read isolation level 7-308

DIRTY READ keywords, syntax in
 SET ISOLATION 7-307

Display label
 syntax in SELECT 7-260
 with UNION operator 7-287

DISTINCT keyword
 syntax
 in CREATE INDEX 7-54
 in expression 7-383
 in SELECT 7-260
 use
 in CREATE INDEX 7-55
 in SELECT 7-261
 no effect in subquery 7-356

Division (/) symbol, arithmetic
 operator 7-371

DOCUMENT keyword
 use in stored procedures 8-8

Documentation notes Intro-22

Documentation, types of
 documentation notes Intro-22
 machine notes Intro-22
 release notes Intro-22

DROP AUDIT statement 7-131

DROP CONSTRAINT keywords
 syntax in ALTER TABLE 7-20
 use in ALTER TABLE 7-35

DROP DATABASE
 statement 7-132

DROP INDEX statement
 syntax 7-134

DROP keyword
 syntax in ALTER TABLE 7-20
 use in ALTER TABLE 7-29

DROP SYNONYM statement 7-137

DROP TABLE statement 7-139

DROP VIEW statement 7-141

Duplicate values in a query 7-261

Dynamic management
 statements 7-11

Dynamic SQL
 allocating memory
 in ESQL/C 6-4
 in ESQL/COBOL 6-13
 using a system descriptor area
 in ESQL/C 6-5
 in ESQL/COBOL 6-13
 using an sqllda pointer structure in
 ESQL/C 6-9

E

Editor, specifying with
 DBEDIT 4-10

Effective checking 7-289

Ellipses (...), wildcard in Condition
 segment 7-353

Environment variable
 default assumptions 4-3
 definition of 4-3
 how to set in Bourne shell 4-4
 how to set in C shell 4-4
 listed by product 4-5

Environment variables Intro-11

Error checking
 continuing after error in stored
 procedure 8-71
 error status with ON
 EXCEPTION 8-68
 exception handling 8-32
 in stored procedures 8-32
 simulating errors 8-34
 SQLCA record (4GL) 5-5
 SQLCA record (ESQL/
 COBOL) 5-10
 sqlca structure (ESQL/C) 5-7
 summary table of SQLCA data
 structure 5-4
 with SYSTEM 8-78

ERROR keyword, in the
 WHENEVER statement 7-337,
 7-341

ESCAPE keyword
 syntax in Condition
 segment 7-346
 use
 in Condition segment 7-352
 with LIKE keyword 7-274,
 7-354
 with MATCHES
 keyword 7-274, 7-354
 with WHERE keyword 7-273

ESQL
 error handling 5-3
 SQL Communications Area 5-3

EXCLUSIVE keyword
 syntax
 in DATABASE 7-101
 in LOCK TABLE 7-204
 use
 in DATABASE 7-103
 in LOCK TABLE 7-205

EXECUTE IMMEDIATE statement
 restricted statement types 7-148
 syntax and usage 7-147
 with concatenation
 operator 7-372

EXECUTE ON keywords
 syntax
 in GRANT 7-175
 in REVOKE 7-247
 use
 in GRANT 7-181

- in REVOKE 7-247
- EXECUTE PROCEDURE statement
 - assigning values with 8-25
 - in FOREACH 8-56
 - using 8-12
- EXECUTE statement
 - parameterizing a statement 7-144
 - syntax 7-142
 - USING DESCRIPTOR
 - clause 7-146
 - USING SQL DESCRIPTOR
 - clause 7-145
 - with concatenation
 - operator 7-372
 - with USING keyword 7-144
- EXISTS keyword
 - beginning a subquery 7-275
 - use in condition subquery 7-356
- EXIT statement
 - exiting a loop 8-27
 - syntax 8-50
- Expression
 - in UPDATE 7-329
 - ordering by 7-285
- Expression segment
 - aggregate expressions 7-393
 - calculated expressions 7-383
 - column expressions 7-373
 - combined expressions 7-398
 - constant expressions 7-376
 - expression types 7-371
 - in SPL expressions 8-24
 - syntax 7-371
- EXTEND function
 - syntax in expression 7-383
 - with DATE, DATETIME and INTERVAL 3-26, 3-27
- Expression checking, specifying with DBANSIWARN 4-7
- EXTENT SIZE keywords 7-94

F

- FETCH statement
 - as affected by CLOSE 7-42
 - checking results with SQLCA 7-161
 - locking for update 7-160

- relation to GET DESCRIPTOR 7-169
- specifying a value's memory location 7-157
- syntax 7-153
- with
 - concatenation operator 7-372
 - INTO keyword 7-266
 - program arrays 7-158
 - scroll cursor 7-155
 - sequential cursor 7-155
- Field qualifier
 - for DATETIME 3-8, 7-368
 - for INTERVAL 3-14, 7-414, 7-420
- File
 - extension
 - .dbs 7-50, 7-102
 - .lok 7-312
 - sending output with the OUTPUT statement 7-216
- FIRST keyword
 - syntax in FETCH 7-153
 - use in FETCH 7-155
- FLOAT data type
 - changing data types 3-22
 - description of 3-12
 - syntax 7-365
 - using as default value 7-24, 7-81
- FLUSH statement
 - syntax 7-162
 - with concatenation
 - operator 7-372
- FOR keyword
 - in CONTINUE 8-40
 - in CREATE AUDIT 7-47
 - in CREATE SYNONYM 7-70
 - in EXIT 8-50
- FOR statement
 - looping in a stored procedure 8-27
 - specifying multiple ranges 8-54
 - syntax 8-52
 - using expression lists 8-54
 - using increments 8-53
- FOR TABLE keywords, in UPDATE STATISTICS 7-335
- FOR UPDATE keywords
 - relation to UPDATE 7-333
 - syntax in DECLARE 7-107

- use
 - in DECLARE 7-111, 7-114, 7-117
 - with column list 7-116
- FOREACH keyword
 - in CONTINUE statement 8-40
 - in EXIT 8-50
- FOREACH statement 7-213
 - looping in a stored procedure 8-27
 - syntax 8-56
- Foreign key 7-27, 7-85, 7-86
- FOREIGN KEY keywords
 - in ALTER TABLE 7-33
 - in CREATE TABLE 7-84
- Format
 - specifying for DATE value with DBDATE 4-8
 - specifying for MONEY value with DBMONEY 4-12
- FRACTION keyword
 - syntax
 - in DATETIME data type 7-368
 - in INTERVAL data type 7-414
 - use
 - as DATETIME field qualifier 3-9, 7-417
 - as INTERVAL field qualifier 3-14, 7-420
- FREE statement
 - effect on BYTE, TEXT variables 7-168
 - effect on cursors 7-214
 - syntax 7-165
 - with concatenation
 - operator 7-372
- FROM keyword
 - syntax
 - in PUT 7-230
 - in REVOKE 7-247
 - in SELECT 7-258
 - use
 - in PUT 7-234
 - in SELECT 7-269
- Function
 - within a stored procedure 8-28
- Function expression, in SELECT 7-262

G

GET DESCRIPTOR statement
 syntax 7-169
 the COUNT keyword 7-171
 with concatenation
 operator 7-372

GOTO keyword, in the
 WHENEVER statement 7-337,
 7-341

GRANT statement
 changing grantor 7-182
 creating a privilege chain 7-182
 database-level privileges 7-177
 default table privileges 7-181
 passing grant ability 7-181
 privileges on a view 7-183
 syntax 7-175
 table-level privileges 7-179
 with CREATE SCHEMA 7-68

GROUP BY keywords
 syntax in SELECT 7-258
 use in SELECT 7-279

H

HAVING keyword
 syntax in SELECT 7-258
 use in SELECT 7-281

Header
 of a procedure 8-29

HEX function, use in
 expression 7-390

HIGH keyword 7-315

Hold cursor
 definition of 7-111
 insert cursor with hold 7-120
 use of 7-112

HOURLY keyword
 syntax
 in DATETIME data type 7-368
 in INTERVAL data type 7-414
 use
 as DATETIME field
 qualifier 3-8, 7-417
 as INTERVAL field
 qualifier 3-14, 7-420

Hyphen (-)
 as delimiter in DATETIME 3-10
 as delimiter in INTERVAL 3-15

I

IBM Informix 4GL
 STATUS variable 5-5
 using SQLCODE with 5-5
 WHENEVER ERROR
 statement 7-337

Icon, explanation of Intro-13

IDATA field
 description of 6-8, 6-17
 with X/Open programs 7-172

IF statement
 branching 8-26
 syntax 8-60
 syntax and use 8-60
 with null values 8-61

ILENGTH field
 description of 6-8, 6-17
 with X/Open programs 7-172

IMMEDIATE keyword, in the SET
 CONSTRAINTS
 statement 7-289

IN keyword
 syntax
 in CREATE AUDIT 7-47
 in CREATE DATABASE 7-49
 in CREATE TABLE 7-92
 in LOCK TABLE 7-204
 use
 in Condition segment 7-350
 in Condition subquery 7-355
 with WHERE keyword 7-273

Index
 displaying information for 7-186
 dropping with DROP
 INDEX 7-134
 naming conventions 7-360, 7-413,
 7-433
 sharing with constraints 7-26,
 7-78
 with temporary tables 7-286

INDEX keyword
 syntax
 in GRANT 7-179
 in REVOKE 7-249
 use
 in GRANT 7-180
 in REVOKE 7-250

Index Name segment
 syntax 7-360, 7-424
 use 7-432

Index privilege 7-180

INDEXES FOR keywords, in INFO
 statement 7-186

INDICATOR field
 description of 6-7, 6-16
 setting with SET
 DESCRIPTOR 7-299

INDICATOR keyword, in
 SELECT 7-265

Indicator variable
 in EXECUTE 7-144
 in expression 7-397

Indicator variables
 in SELECT 7-265

Industry standards, compliance
 with Intro-22

INFO statement
 displaying privileges and
 status 7-187
 displaying tables, columns, and
 indexes 7-186
 syntax 7-185

Informix extension checking,
 specifying with
 DBANSIWARN 4-7

INFORMIXCOB environment
 variable 4-20

INFORMIXCOBDIR environment
 variable 4-20

INFORMIXCOBSTORE
 environment variable 4-21

INFORMIXCOBTYPE environment
 variable 4-21

INFORMIXDIR environment
 variable 4-22

INFORMIXTERM environment
 variable 4-23

informix, privileges associated with
 user 7-178

Insert buffer
 counting inserted rows 7-164,
 7-237

filling with constant values 7-232
 inserting rows with a
 cursor 7-192
 storing rows with PUT 7-231
 triggering flushing 7-235
Insert cursor
 closing 7-42
 definition of 7-110
 in INSERT 7-192
 in PUT 7-233
 opening 7-209
 reopening 7-210
 result of CLOSE in SQLCA 7-42
 use of 7-111
 with hold 7-120
 writing buffered rows with
 FLUSH 7-162
INSERT INTO keywords
 in INSERT 7-190
 in LOAD 7-203
INSERT keyword
 syntax
 in GRANT 7-179
 in REVOKE 7-249
 use
 in GRANT 7-180
 in REVOKE 7-250
INSERT statement
 effect of transactions 7-193
 filling insert buffer with
 PUT 7-231
 in dynamic SQL 7-197
 inserting
 rows through a view 7-191
 rows with a cursor 7-192
 values into SERIAL
 columns 3-18
 SERIAL columns 7-196
 specifying values to insert 7-194
 syntax 7-189
 use with insert cursor 7-120
 with
 DECLARE 7-107
 SELECT 7-197
Installation directory, specifying
 with INFORMIXDIR 4-22
INTEGER data type
 changing data types 3-22
 description of 3-13

syntax 7-365
 using as default value 7-24, 7-81
Intensity attributes, setting
 INFORMIXTERM for 4-23
INTERVAL data type
 adding or subtracting from 3-30
 adding or subtracting from
 DATETIME values 3-27
 as quoted string 7-427
 description of 3-13
 field delimiters 3-15
 field qualifier, syntax 7-414
 in expression 7-381
 in INSERT 7-428
 multiplying or dividing
 values 3-30
 range of expressions 3-25
 range of operations with DATE
 and DATETIME 3-25
 syntax 7-365, 7-420
 with EXTEND function 3-26, 3-27
INTERVAL Field Qualifier
 segment 7-414
INTO keyword
 in SELECT 7-265
 syntax
 in FETCH 7-153
 in SELECT 7-258
 use
 in FETCH 7-158
 in SELECT 7-265
INTO TEMP keywords
 syntax in SELECT 7-258
 use
 in SELECT 7-285
 with UNION operator 7-287
IS keyword
 in Condition segment 7-351
 with WHERE keyword 7-273
IS NOT keywords, syntax in
 Condition segment 7-346
IS NULL keywords 7-273
Isolation level
 Committed Read 7-308
 Cursor Stability 7-309
 definitions 7-308
 Dirty Read 7-308
 in external tables 7-309, 7-312
 Repeatable Read 7-309

items table in stores5 database,
 columns in 1-6
ITYPE field
 description of 6-8, 6-17
 setting with SET
 DESCRIPTOR 7-299
 with X/Open programs 7-172

J

Join
 in Condition segment 7-277
 multiple-table join 7-278
 outer join 7-278
 self-join 7-278
 two-table join 7-277

L

LAST keyword
 syntax in FETCH 7-153
 use in FETCH 7-155
LENGTH field
 description of 6-7, 6-16
 setting with SET
 DESCRIPTOR 7-298
 with DATETIME and INTERVAL
 types 7-299
 with DECIMAL and MONEY
 types 7-298
LENGTH function
 in expression 7-262
 syntax in expression 7-383
 use in expression 7-389
LET statement
 assigning values 8-25
 executing a procedure 8-12
 syntax 8-64
LIKE keyword
 syntax in Condition
 segment 7-346
 use
 in Condition segment 7-352
 in SELECT 7-273
 wildcard characters 7-274
Literal
 DATETIME
 in Condition segment 7-346

- in expression 7-376, 7-381
 - segment 7-416
 - syntax 7-417
 - syntax in INSERT 7-194
 - use in ALTER TABLE 7-23
 - use in CREATE TABLE 7-81
 - with IN keyword 7-273
 - DATE, using as a default value 7-24, 7-82
 - INTERVAL
 - in Condition segment 7-346
 - in expression 7-376, 7-381
 - segment 7-419
 - syntax 7-420
 - syntax in INSERT 7-194
 - using as default value 7-24, 7-82
 - Number
 - in Condition segment 7-346
 - in expression 7-376, 7-379
 - segment 7-422
 - syntax 7-422
 - syntax in INSERT 7-194
 - with IN keyword 7-351
 - LOAD statement
 - DELIMITER clause 7-203
 - input formats for data 7-201
 - INSERT INTO clause 7-203
 - loading VARCHAR, TEXT, or BYTE data types 7-202
 - specifying field delimiter with DBDELIMITER 4-9
 - specifying the table to load into 7-203
 - syntax 7-199
 - the LOAD FROM file 7-200
 - LOCK MODE keywords
 - syntax
 - in ALTER TABLE 7-20
 - in CREATE TABLE 7-95
 - use
 - in ALTER TABLE 7-36
 - in CREATE TABLE 7-95
 - LOCK TABLE statement
 - in databases with transactions 7-205
 - in databases without transactions 7-206
 - syntax 7-204
 - Locking
 - during
 - delete 7-122
 - inserts 7-193
 - updates 7-115, 7-327
 - overriding row-level 7-205
 - releasing with COMMIT WORK 7-46
 - releasing with ROLLBACK WORK 7-254
 - types of locks
 - page lock 7-95
 - row lock 7-95
 - update cursors effect on 7-115
 - waiting period 7-311
 - with
 - FETCH 7-160
 - scroll cursor 7-309
 - SET ISOLATION 7-307
 - SET LOCK MODE 7-311
 - UNLOCK TABLE 7-323
 - within transaction 7-37
 - LOG IN keywords, syntax in CREATE DATABASE 7-49
 - Logging
 - buffered vs. unbuffered 7-313
 - changing mode with SET LOG 7-313
 - finding log file location 7-53
 - renaming log 7-318
 - setting with CREATE TABLE 7-91
 - starting with START DATABASE 7-52, 7-317
 - with IBM Informix OnLine 7-51
 - with IBM Informix SE 7-52
 - Logical operator
 - in Condition segment 7-358
 - .lok extension 7-312
 - Loop
 - controlled 8-52
 - creating and exiting in SPL 8-27
 - exiting using RAISE exception 8-35
 - indefinite with WHILE 8-84
 - LOW keyword 7-315
-
- ## M
- Machine notes Intro-22
 - manufact table in stores5 database, columns in 1-11
 - mary 7-32
 - MATCHES keyword
 - syntax in Condition segment 7-346
 - use
 - in Condition segment 7-352
 - in SELECT 7-273
 - wildcard characters 7-274
 - MAX function
 - syntax in expression 7-383
 - use in expression 7-395
 - MDY function
 - syntax in expression 7-383
 - Memory
 - allocating for a system sqlda structure 7-13
 - allocating in ESQL/C 6-4
 - allocating in ESQL/COBOL 6-13
 - Memory, releasing with FREE 7-165
 - Menu, specifying with DBMENU 4-12
 - Message file for error messages Intro-19
 - Message files, specifying subdirectory with DBLANG 4-11
 - MIN function
 - syntax in expression 7-383
 - use in expression 7-395
 - Minus (-) sign, arithmetic operator 7-371
 - MINUTE keyword
 - syntax
 - in DATETIME data type 7-368
 - in INTERVAL data type 7-414
 - use
 - as DATETIME field qualifier 3-8, 7-417
 - as INTERVAL field qualifier 3-14, 7-420
 - MODE ANSI keywords
 - syntax
 - in CREATE DATABASE 7-49

- in START DATABASE 7-317
- use
 - in CREATE DATABASE 7-52
 - in START DATABASE 7-318
- MODIFY keyword
 - syntax in ALTER TABLE 7-20
 - use in ALTER TABLE 7-30
- MODIFY NEXT SIZE keywords
 - syntax in ALTER TABLE 7-20
 - use in ALTER TABLE 7-36
- MONEY data type
 - changing data types 3-22
 - description of 3-17
 - syntax 7-365
 - using as default value 7-24, 7-81
- MONEY value
 - setting DBMONEY environment variable 4-12
 - specifying European format with DBMONEY 4-13
- MONTH function
 - syntax in expression 7-383
 - use in expression 7-385
- MONTH keyword
 - syntax
 - in DATETIME data type 7-368
 - in INTERVAL data type 7-414
 - use
 - as DATETIME field
 - qualifier 3-8, 7-417
 - as INTERVAL field
 - qualifier 3-14, 7-420
- Multi-row query
 - destination of returned values 7-157
 - managing with FETCH 7-154

N

- NAME field, description of 6-7, 6-16
- Naming convention
 - column 7-80, 7-241
 - database 7-363
 - index 7-360, 7-413, 7-433
 - table 7-76, 7-400, 7-435
- Nested ordering, in SELECT 7-284

- Network environment variable
 - DBNETTYPE 4-13
 - DBPATH 4-14
 - SQLRM 4-26
 - SQLRMDIR 4-27
- NEXT keyword
 - syntax in FETCH 7-153
 - use in FETCH 7-155
- NEXT SIZE keywords
 - use in CREATE TABLE 7-94
 - use in GRANT 7-178
- NOSORTINDEX environment variable 4-24
- NOT CLUSTER keywords
 - syntax in ALTER INDEX 7-17
 - use in ALTER TABLE 7-19
- NOT FOUND keywords, in the WHENEVER statement 7-337, 7-340
- NOT IN keywords, use in Condition subquery 7-355
- NOT keyword
 - syntax
 - in Condition segment 7-345, 7-346
 - with BETWEEN keyword 7-272
 - with IN keyword 7-275
 - use
 - in Condition segment 7-352
 - with LIKE, MATCHES keywords 7-273
- NOT NULL keywords
 - syntax
 - in ALTER TABLE 7-22
 - in CREATE TABLE 7-80
 - use
 - in ALTER TABLE 7-30
 - in CREATE TABLE 7-83
 - with IS keyword 7-273
- NOT WAIT keywords, in SET LOCK MODE 7-311
- NOTFOUND keyword, contrasted with NOT FOUND keywords 7-340
- NULL keyword, ambiguous as procedure variable 7-409
- Null value
 - checking for in SELECT 7-265
 - in IF statement 8-61

- restrictions on primary key 7-85
- returned implicitly by stored procedure 8-75
- specifying as default value 7-25
- with WHILE statement 8-84
- NULLABLE field, description of 6-8, 6-17

O

- OF keyword
 - syntax in DECLARE 7-107
 - use in DECLARE 7-116
- ON EXCEPTION statement
 - placement of 8-69
 - scope of control 8-33
 - syntax 8-67
 - trapping errors 8-32
 - user-generated errors 8-34
- ON keyword
 - syntax
 - in CREATE INDEX 7-54
 - in GRANT 7-175
 - in REVOKE 7-247
 - use
 - in CREATE INDEX 7-55
 - in GRANT 7-181
- Online
 - files Intro-22
 - help Intro-22
- OPEN statement
 - constructing the active set 7-208
 - opening an insert cursor 7-209
 - opening select or update cursors 7-208
 - reopening a cursor 7-210
 - substituting values for ? parameters 7-211
 - syntax 7-207
 - when to use FOREACH 7-213
 - with concatenation operator 7-372
 - with FREE 7-214
- Optimization, specifying a high or low level 7-315
- Optimizer
 - and SET OPTIMIZATION statement 7-315

Optimizing
 a query 7-301
 a server 7-315
 across a network 7-316
 updating system catalog
 tables 7-335

OR keyword
 syntax in Condition
 segment 7-345
 use in Condition segment 7-358

ORDER BY keywords
 ascending order 7-284
 descending order 7-284
 restrictions in INSERT 7-197
 select columns by number 7-285
 syntax in SELECT 7-258
 use
 in SELECT 7-283
 with UNION operator 7-287

orders table in stores5 database,
 columns in 1-6

OUTER keyword, with FROM
 keyword in SELECT 7-269

OUTPUT statement, syntax and
 use 7-216

Owner
 in ALTER TABLE 7-21
 in CREATE SYNONYM 7-71
 in Index Name segment 7-360,
 7-413, 7-424, 7-433
 in RENAME COLUMN 7-241
 in RENAME TABLE 7-243
 in Table Name segment 7-399,
 7-435
 in View Name segment 7-438
 of view in CREATE VIEW 7-439

Owner-privileged procedure 8-16

P

PAGE keyword
 use in ALTER TABLE 7-36
 use in CREATE TABLE 7-95

Parameter
 BYTE or TEXT in SPL 8-49
 in CALL statement 8-38
 to a stored procedure 8-29

Parameterizing a statement
 in PREPARE 7-223
 with SQL identifiers 7-225

Parent-child relationship 7-27, 7-86

PATH environment variable 4-28

Pathname
 including in SQLEXEC 4-25
 specifying with DBPATH 4-14
 specifying with DBSRC 4-16
 specifying with PATH 4-28

Percent (%) sign, wildcard in
 Condition segment 7-353

PERFORM keyword, in the
 WHENEVER statement 7-337

Performance
 increasing with stored
 procedures 8-6

Permission
 with SYSTEM 8-79

Phantom row 7-308

PIPE keyword, in the OUTPUT
 statement 7-217

Plus (+) sign, arithmetic
 operator 7-371

PRECISION field
 description of 6-7, 6-16
 setting with SET
 DESCRIPTOR 7-298
 with GET DESCRIPTOR 7-173

PREPARE statement
 executing 7-142
 increasing performance
 efficiency 7-229
 multi-statement text 7-222, 7-228
 parameterizing a statement 7-223
 parameterizing for SQL
 identifiers 7-225
 question (?) mark as
 placeholder 7-219
 releasing resources with
 FREE 7-167
 restrictions with SELECT 7-221
 statement identifier use 7-219
 syntax 7-218
 valid statement text 7-221
 with concatenation
 operator 7-372

Prepared statement
 describing returned values with
 DESCRIBE 7-126
 executing 7-142
 prepared object limit 7-219
 valid statement text 7-221

PREVIOUS keyword
 syntax in FETCH 7-153
 use in FETCH 7-155

Primary key constraint
 data type conversion 7-32
 defining column as 7-85
 dropping 7-35
 enforcing 7-78
 modifying a column with 7-31
 referencing 7-27
 requirements for 7-26, 7-85
 rules of use 7-34, 7-86

PRIMARY KEY keywords
 in ALTER TABLE statement 7-25
 in CREATE TABLE 7-83, 7-84
 use in ALTER TABLE 7-33

Printing, specifying print program
 with DBPRINT 4-15

PRIOR keyword
 syntax in FETCH 7-153
 use in FETCH 7-155

Privilege
 Alter 7-180
 Connect 7-177
 DBA 7-178
 default for stored
 procedures 8-16
 default for table using CREATE
 TABLE 7-76
 Delete 7-180
 displaying with the INFO
 statement 7-187
 encoded in system catalog 2-29,
 2-32
 Index 7-180
 Insert 7-180
 needed
 to create a view 7-183
 to drop an index 7-134
 to modify data 7-180
 on a synonym 7-70
 on a view 7-98
 Resource 7-178

Update 7-180
 when privileges conflict 7-176
 with DBA-privileged procedures 8-18
 with owner-privileged procedures 8-17
 PRIVILEGES FOR keywords, in INFO statement 7-187
 PRIVILEGES keyword
 syntax
 in GRANT 7-179
 in REVOKE 7-249
 use
 in GRANT 7-180
 in REVOKE 7-250
 Procedure name
 conflict with function name 7-424
 naming conventions 7-424
 PUBLIC keyword
 syntax
 in GRANT 7-175
 in REVOKE 7-247
 use
 in GRANT 7-178
 in REVOKE 7-249
 PUT statement
 source of row values 7-232
 use in transactions 7-231
 with concatenation operator 7-372
 with FLUSH 7-231

Q

Qualifier, field
 for DATETIME 3-8, 7-368, 7-417
 for INTERVAL 3-14, 7-414, 7-420
 Query
 piping results to another program 7-217
 sending results to an operating system file 7-216
 sending results to another program 7-217
 Query optimization information statements 7-11
 Question (?) mark
 as placeholder in PREPARE 7-219

naming variables in PUT 7-234
 replacing with USING keyword 7-211
 supplying values to placeholders with EXECUTE 7-144
 wildcard in condition 7-353
 Quoted string
 in expression 7-376
 syntax
 in Condition segment 7-346
 in expression 7-383
 in INSERT 7-194
 use
 in expression 7-377
 in INSERT 7-428
 with LIKE, MATCHES keywords 7-273
 Quoted String segment
 DATETIME, INTERVAL values as strings 7-427
 syntax 7-426
 wildcards 7-427
 with LIKE in a condition 7-427

R

RAISE EXCEPTION statement
 exiting a loop 8-27
 syntax 8-73
 RECOVER TABLE statement
 archiving a database with audit trails 7-238
 manipulating audit trail file 7-239
 syntax 7-238
 Recursion, in a stored procedure 8-29
 REFERENCES FOR keywords, in INFO statement 7-187
 REFERENCES keyword
 in ALTER TABLE 7-27
 in CREATE TABLE 7-86, 7-88
 syntax
 in GRANT 7-179
 in REVOKE 7-249
 use
 in GRANT 7-180
 in REVOKE 7-250
 References privilege
 definition of 7-180
 displaying with the INFO statement 7-187
 Referential constraint
 data type restrictions 7-88
 definition of 7-27, 7-86
 dropping 7-35
 enforcing 7-78
 modifying a column with 7-31
 rules of use 7-86
 Relational Operator
 segment 7-429
 Relational operator
 in Condition segment 7-346
 with WHERE keyword in SELECT 7-272
 Relational Operator segment 7-429
 RELATIVE keyword
 syntax in FETCH 7-153
 use in FETCH 7-156
 Relay Module
 SQLRM environment
 variable 4-26
 SQLRMDIR environment
 variable 4-27
 Release notes Intro-22
 RENAME COLUMN statement
 restrictions 7-241
 syntax 7-241
 RENAME TABLE statement
 ANSI-compliant naming 7-243
 syntax 7-243
 REPAIR TABLE statement, syntax and use 7-245
 Repeatable Read isolation level
 description of 7-309
 emulating during update 7-160
 REPEATABLE READ keywords, syntax in SET ISOLATION 7-307
 RESOURCE keyword
 use in GRANT 7-178
 use in REVOKE 7-251
 Resource privilege 7-178
 Resource privilege with SPL 8-16
 RETURN statement
 exiting a loop 8-27
 returning insufficient values 8-75

returning null values 8-75
 syntax 8-75
 REVOKE statement
 column-specific privileges 7-251
 database-level privileges 7-251
 privileges needed 7-248
 syntax 7-247
 table-level privileges 7-249
 ROLLBACK WORK statement
 restriction with FOREACH 7-255
 syntax 7-254
 use with WHENEVER 7-38, 7-45, 7-255
 ROLLFORWARD DATABASE
 statement
 exclusive locking 7-256
 syntax 7-256
 ROUND function, use in
 expression 7-391
 Row
 deleting 7-122
 engine response to locked
 row 7-311
 inserting through a view 7-191
 inserting with a cursor 7-192
 multi-row queries with
 FETCH 7-154
 phantom 7-308
 retrieving with FETCH 7-156
 rowid definition 7-156
 updating through a view 7-326
 writing buffered rows with
 FLUSH 7-162
 ROW keyword
 use in ALTER TABLE 7-36
 use in CREATE TABLE 7-95
 Rowid 7-405
 Run-time program
 setting DBANSIWARN 4-7
 setting INFORMIXCOBDIR 4-20

S

SCALE field
 description of 6-7, 6-16
 setting with SET
 DESCRIPTOR 7-298
 with GET DESCRIPTOR 7-173
 Scroll cursor
 definition of 7-111
 use of 7-112
 with FETCH 7-155
 with hold, in a transaction 7-309
 SCROLL keyword
 syntax in DECLARE 7-107
 use in DECLARE 7-112
 SECOND keyword
 syntax
 in DATETIME data type 7-368
 in INTERVAL data type 7-414
 use
 as DATETIME field
 qualifier 3-9, 7-417
 as INTERVAL field
 qualifier 3-14, 7-420
 Select cursor
 closing 7-42
 definition of 7-110
 opening 7-208
 reopening 7-210
 use of 7-110
 SELECT keyword
 ambiguous use as procedure
 variable 7-409
 syntax
 in GRANT 7-179
 in REVOKE 7-249
 use
 in GRANT 7-180
 in REVOKE 7-250
 Select privilege
 definition of 7-180
 SELECT statement
 aggregate functions in 7-393
 as an argument to a stored
 procedure 8-38
 assigning values with 8-25
 associating with cursor with
 DECLARE 7-110
 column numbers 7-281, 7-285
 describing returned values with
 DESCRIBE 7-125
 FROM Clause 7-269
 GROUP BY clause 7-279
 HAVING clause 7-281
 INTO clause with I4GL,
 ESQL 7-265
 INTO TEMP clause 7-285
 joining tables in WHERE
 clause 7-277
 ORDER BY clause 7-283
 remote query 7-259
 restrictions with INTO
 clause 7-221
 SELECT clause 7-260
 select numbers 7-281, 7-285
 subquery with WHERE
 keyword 7-272
 syntax 7-258
 UNION operator 7-287
 use of expressions 7-261
 with
 Condition segment 7-345
 DECLARE 7-107
 FOREACH 8-56
 INSERT 7-197
 INTO keyword 7-157
 LET 8-65
 writing rows retrieved to an
 ASCII file 7-319
 Self-join
 description of 7-278
 Sequential cursor
 definition of 7-111
 use of 7-111
 with FETCH 7-155
 SERIAL data type
 description of 3-18
 in ALTER TABLE 7-22
 in INSERT 7-196
 inserting values 3-18
 resetting values 3-18
 syntax 7-365
 with stored procedures 8-43
 SET CONSTRAINTS statement,
 syntax and use 7-289
 SET DEBUG FILE statement
 with TRACE 8-80
 SET DEBUG FILE TO statement,
 syntax and use 7-291
 SET DESCRIPTOR statement
 syntax 7-293
 the VALUE option 7-296
 SET EXPLAIN statement
 interpreting output 7-301
 MERGE JOIN information 7-303

- optimizer access paths 7-302
- output examples 7-303
- SORT SCAN information 7-303
 - syntax 7-301
- SET ISOLATION statement
 - default database levels 7-307
 - definition of isolation levels 7-308
 - effects of isolation 7-309
 - syntax 7-307
- SET keyword
 - syntax in UPDATE 7-325
 - use in UPDATE 7-328
- SET LOCK MODE statement
 - kernel locking 7-312
 - setting wait period 7-312
 - syntax 7-311
- SET LOG statement
 - buffered vs. unbuffered 7-313
 - syntax 7-313
- SET OPTIMIZATION statement,
 - syntax and use 7-315
- Setting environment variables 4-4
- SHARE keyword, syntax in LOCK TABLE 7-204
- Shared memory parameters,
 - specifying file with TBCONFIG 4-27
- Simple assignment 8-64
- Single-precision floating-point
 - number, storage of 3-12
- SITENAME function
 - returns servername 7-378
 - syntax
 - in expression 7-376
 - in INSERT 7-194
 - use
 - in ALTER TABLE 7-23
 - in CREATE TABLE 7-81
 - in expression 7-378
 - in INSERT 7-196
- Slash (/), arithmetic operator 7-371
- SMALLFLOAT data type
 - changing data types 3-22
 - description of 3-19
 - syntax 7-365
- SMALLINT data type
 - changing data types 3-22
 - description of 3-19
 - syntax 7-365
 - using as default value 7-24, 7-81
- SOME keyword
 - beginning a subquery 7-276
 - use in Condition subquery 7-357
- Sorting
 - in a combined query 7-287
 - in SELECT 7-283
- Space ()
 - as delimiter in DATETIME 3-10
 - as delimiter in INTERVAL 3-15
- SPL
 - flow control statements 8-26
 - relation to SQL 8-5
 - statement syntax 8-36
- SQL
 - statement types 7-9
- SQL Communications Area (SQLCA)
 - description of 5-3
 - effect of setting
 - DBANSIWARN 4-7
 - result after CLOSE 7-42
 - result after DATABASE 7-101
 - result after DESCRIBE 7-126
 - result after FETCH 7-161
 - result after FLUSH 7-163
 - result after OPEN 7-209
 - result after PUT 7-236
 - result after SELECT 7-268
- SQLCA.SQLAWARN (4GL) 5-6
 - sqlca.sqlcode
 - in ESQL/C 5-8
 - SQLCA.SQLCODE (4GL) 5-5
 - sqlca.sqlerrd
 - in ESQL/C 5-8
 - SQLCA.SQLERRD (4GL) 5-5
 - sqlca.sqlwarn
 - in ESQL/C 5-10
 - SQLCODE OF SQLCA (ESQL/COBOL) 5-11
- sqllda structure
 - fields in sqllda 6-11
 - fields in sqlvar_struct 6-11
 - sqllda.h header file shown 6-10
 - syntax
 - in DESCRIBE 7-125
 - in EXECUTE 7-142
 - in FETCH 7-153
 - in OPEN 7-208
 - in PUT 7-230
- system descriptor area in ESQL/C 6-5
- system descriptor area in ESQL/COBOL 6-13
 - use
 - in DESCRIBE 7-127
 - in FETCH 7-159
 - in OPEN 7-213
 - in PUT 7-234
 - using pointers in ESQL/C 6-9
- sqllda.h header file
 - fields in sqllda 6-11
 - fields in sqlvar_struct 6-11
 - shown 6-10
- SQLERRD OF SQLCA (ESQL/COBOL) 5-12
- SQLERROR keyword, in the WHENEVER statement 7-337
- SQLEXEC environment
 - variable 4-25
- SQLRM environment variable 4-26
- SQLRMDIR environment
 - variable 4-27
- SQLWARN of SQLWARN (ESQL/COBOL) 5-13
- SQLWARNING keyword, in the WHENEVER statement 7-340
- START DATABASE statement
 - syntax and use 7-317
- state table in stores5 database,
 - columns in 1-11
- Statement identifier
 - associating with cursor 7-110
 - definition of 7-219
 - releasing 7-220
 - syntax
 - in DECLARE 7-107
 - in DESCRIBE 7-125
 - in EXECUTE 7-142
 - in FREE 7-166
 - in PREPARE 7-218
 - use
 - in DECLARE 7-117
 - in FREE 7-167
 - in PREPARE 7-219
- Statement types 7-9
- Statement variable name,
 - definition 7-147

- STATUS FOR keywords, in INFO statement 7-188
 - STATUS variable (4GL) 5-5
 - Status, displaying with INFO statement 7-188
 - stock table in stores5 database, columns in 1-8
 - STOP keyword, in the WHENEVER statement 7-337, 7-342
 - Stored procedure
 - altering 8-16
 - BYTE and TEXT data types 8-44, 8-48
 - comments in 8-8
 - creating from an embedded language 8-8
 - creating from DBACCESS 8-7
 - cursors with 8-56
 - DBA-privileged 8-16
 - debugging 8-14, 8-80
 - definition of 8-6
 - displaying contents 8-11
 - displaying documentation 8-11
 - documenting 8-8
 - executing 8-12
 - granting privileges on 7-181
 - handling multiple rows 8-76
 - header 8-29, 8-43
 - introduction to 8-5
 - name confusion with SQL functions 8-23
 - naming output file for TRACE statement 7-291
 - owner-privileged 8-16
 - receiving data from SELECT 7-265
 - recursion 8-29
 - returning values 8-29
 - revoking privileges on 7-248
 - simulating errors 8-73
 - use 8-6
 - variable 8-19
 - stores5 database
 - call_type columns 1-10
 - catalog table columns 1-9
 - copying Intro-7
 - creating on IBM Informix OnLine Intro-7
 - creating on IBM Informix SE Intro-8
 - customer table columns 1-5
 - cust_calls table columns 1-10
 - data values 1-21
 - description of 1-3
 - items table columns 1-6
 - manufact table columns 1-11
 - map of 1-11
 - orders table columns 1-6
 - overview Intro-6
 - primary-foreign key relationships 1-13 to 1-21
 - state table columns 1-11
 - stock table columns 1-8
 - structure of tables 1-4
 - Subquery
 - beginning with ALL/ANY/SOME keywords 7-276
 - beginning with EXISTS keyword 7-275
 - beginning with IN keyword 7-275
 - correlated 7-355
 - definition of 7-272
 - in Condition segment 7-355
 - restrictions with UNION operator 7-287
 - with DISTINCT keyword 7-261
 - SUM function
 - syntax in expression 7-383
 - use in expression 7-396
 - Synonym
 - ANSI-compliant naming 7-71
 - chaining 7-73
 - creating with CREATE SYNONYM 7-70
 - difference from alias 7-70
 - dropping 7-137
 - Syntax diagram
 - conventions Intro-12
 - elements of Intro-15
 - System catalog
 - accessing 2-8
 - database entries 7-50
 - description of 2-3
 - map of tables 2-33
 - sysblobs 2-10
 - syschecks 2-11
 - syscolauth 2-11, 7-249
 - syscoldepend 2-12
 - syscolumns 2-13
 - sysconstraints 2-16
 - sysdefaults 2-17
 - sysdepend 2-18
 - sysindexes 2-18
 - sysindexes entry for constraint 7-78
 - sysopclstr 2-21
 - sysprocauth 2-23
 - sysprocbody 2-24, 8-11
 - sysprocedures 2-25
 - sysprocplan 2-26
 - sysreferences 2-27
 - sys synonyms 2-27
 - sys syntable 2-28
 - ystabauth 2-29, 7-183, 7-249
 - ystables 2-30
 - sysusers 2-32
 - sysviews 2-33
 - updating 2-9
 - updating data for optimization 7-335
 - updating statistics 2-9
 - System descriptor area
 - fields defined 6-7, 6-16
 - in ESQL/C 6-5
 - in ESQL/COBOL 6-13
 - modifying contents 7-293
 - resizing 7-295
 - values for TYPE and ITYPE fields 6-8, 6-17
 - System name, in database name 7-363
 - SYSTEM statement
 - syntax 8-78
-
- T**
- Table
 - adding a constraint 7-33
 - alias in SELECT 7-269
 - ANSI-compliant naming 7-435
 - changing the data type of a column 3-22
 - checking with the CHECK TABLE statement 7-39

- creating
 - a synonym for 7-70
 - a table 7-75
 - a temporary table 7-90
 - dropping
 - a constraint 7-35
 - a synonym 7-137
 - a table 7-139
 - engine response to locked table 7-311
 - joins in Condition segment 7-277
 - loading data with the LOAD statement 7-199
 - locking
 - changing mode 7-36
 - with ALTER INDEX 7-18
 - with LOCK TABLE 7-204
 - logging 7-91
 - naming conventions 7-76, 7-400, 7-435
 - optimizing queries 7-336
 - renaming 7-243
 - repairing with REPAIR TABLE statement 7-245
 - restoring with audit trail 7-238
 - structure in stores5 database 1-4
 - system catalog tables 2-10 to 2-33
 - unlocking 7-323
 - use of temporary 7-91
 - TABLE keyword, syntax in UPDATE STATISTICS 7-335
 - Table Name segment 7-434
 - Table-level privilege
 - column-specific privileges 7-251
 - default with GRANT 7-181
 - definition and use 7-180
 - granting 7-179
 - passing grant ability 7-181
 - revoking 7-249
 - TABLES keyword, in INFO statement 7-186
 - TBCONFIG environment variable 4-27
 - tbconfig file, specifying with TBCONFIG 4-27
 - TEMP keyword
 - syntax in SELECT 7-258
 - use in SELECT 7-285
 - TEMP TABLE keywords, syntax in CREATE TABLE 7-75
 - Temporary files, specifying directory with DBTEMP 4-17
 - Temporary table
 - creating constraints for 7-91
 - TERM environment variable 4-29
 - TERMCAP environment variable 4-29
 - termcap file
 - and TERMCAP environment variable 4-29
 - selecting with INFORMIXTERM 4-24, 4-25
 - Terminal handling and TERM environment variable 4-29
 - and TERMCAP environment variable 4-29
 - and TERMINFO environment variable 4-30
 - terminfo directory and TERMINFO environment variable 4-30
 - selecting with INFORMIXTERM 4-24
 - TERMINFO environment variable 4-30
 - TEXT data type
 - description of 3-19
 - requirements for LOAD statement 7-202
 - selecting a column 3-20
 - syntax 7-365
 - with control characters 3-20
 - with stored procedures 8-44, 8-48
 - Text editor, specifying with DBEDIT 4-10
 - Time function
 - restrictions with GROUP BY 7-280
 - syntax in expression 7-383
 - use in SELECT 7-262
 - TO CLUSTER keywords, in ALTER INDEX 7-18
 - TO keyword
 - in expression 7-383
 - in GRANT 7-175
 - TODAY function
 - syntax
 - in Condition segment 7-346
 - in expression 7-376
 - in INSERT 7-194
 - use
 - in ALTER TABLE 7-23
 - in constant expression 7-379
 - in CREATE TABLE 7-81
 - in INSERT 7-196
 - TRACE statement
 - debugging a stored procedure 8-14
 - syntax 8-80
 - Transaction and CREATE DATABASE 7-52
 - committing with COMMIT WORK 7-46
 - logging 7-317
 - recovering transactions 7-256
 - rolling back 7-254
 - scroll cursor and data consistency 7-309
 - starting with BEGIN WORK 7-37
 - using cursors in 7-118
 - Transaction logging
 - renaming log 7-318
 - TRUNC function, use in expression 7-392
 - TYPE field
 - changing from BYTE or TEXT 7-299
 - description of 6-7, 6-16
 - setting in SET DESCRIPTOR 7-296
 - setting in X/Open programs 7-298
 - with X/Open programs 7-172
 - Typographical conventions Intro-11
-
- U**
- Underscore (_), wildcard in Condition segment 7-353
 - UNION operator
 - restrictions on use 7-287
 - syntax in SELECT 7-258

- use in SELECT 7-287
- Unique constraint
 - data type conversion 7-32
 - dropping 7-35
 - enforcing 7-78
 - modifying a column with 7-31
 - requirements for 7-25
 - rules of use 7-34, 7-85, 7-86
- UNIQUE keyword
 - syntax
 - in CREATE INDEX 7-54
 - in CREATE TABLE 7-83
 - in SELECT 7-260
 - use
 - in ALTER TABLE 7-33
 - in CREATE INDEX 7-55
 - in CREATE TABLE 7-84
 - in expression 7-393
 - in SELECT 7-261
 - no effect in subquery 7-356
- UNITS keyword
 - syntax in expression 7-376
 - use in expression 7-382
- UNLOAD statement
 - DELIMITER clause 7-322
 - specifying field delimiter with DBDELIMITER 4-9
 - syntax 7-319
 - UNLOAD TO file 7-320
 - unloading VARCHAR, TEXT, or BYTE columns 7-321
- UNLOAD TO file 7-320
- UNLOCK TABLE statement, syntax and use 7-323
- Updatable view 7-100
- Update cursor
 - definition of 7-110
 - locking considerations 7-115
 - opening 7-208
 - restricted statements 7-115
 - use in UPDATE 7-333
 - using 7-114
- UPDATE keyword
 - syntax
 - in GRANT 7-179
 - in REVOKE 7-249
 - use
 - in GRANT 7-180
 - in REVOKE 7-250

- Update privilege
 - definition of 7-180
 - with a view 7-326
- UPDATE statement
 - and transactions 7-326
 - locking considerations 7-327
 - restrictions on columns for
 - update 7-116
 - rolling back updates 7-327
 - syntax 7-325
 - updating through a view 7-326
 - updating with cursor 7-333
 - use of expressions 7-330
 - with
 - Condition segment 7-345
 - FETCH 7-160
 - SET keyword 7-328
 - WHERE CURRENT OF keywords 7-333
 - WHERE keyword 7-332
 - with an update cursor 7-114
- UPDATE STATISTICS statement
 - optimizing search strategies 7-335
 - syntax 7-335
 - when to execute 7-336
- USER function
 - as affected by ANSI compliance 7-378
 - syntax
 - in Condition segment 7-346
 - in expression 7-376
 - in INSERT 7-194
 - use
 - in ALTER TABLE 7-23
 - in CREATE TABLE 7-81
 - in expression 7-377
 - in INSERT 7-196
- User informix, privileges associated with 7-178
- USING DESCRIPTOR keywords
 - information from DESCRIBE 7-127
 - syntax
 - in EXECUTE 7-142
 - in FETCH 7-153
 - in OPEN 7-207
 - in PUT 7-230

- use
 - in FETCH 7-159
 - in OPEN 7-213
 - in PUT 7-146, 7-234, 7-235
- USING keyword
 - syntax
 - in EXECUTE 7-144
 - in OPEN 7-207
 - use
 - in EXECUTE 7-144
 - in OPEN 7-211
- USING SQL DESCRIPTOR keywords
 - in DESCRIBE 7-128
 - in EXECUTE 7-145

V

- VALUE field
 - after NULL value is fetched 7-174
 - use in GET DESCRIPTOR 7-172
- VALUE keyword
 - relation to FETCH 7-173
 - use in SET DESCRIPTOR 7-296
- VALUES keyword
 - syntax in INSERT 7-190
 - use
 - effect with PUT 7-233
 - in INSERT 7-194
- VARCHAR data type
 - considerations for UNLOAD statement 7-321
 - description of 3-21
 - requirements for LOAD statement 7-202
 - syntax 7-365
 - using as default value 7-24, 7-82
- Variable
 - default values in SPL 8-46, 8-47
 - define in SPL 8-42
 - GLOBAL and LOCAL in SPL 8-20
 - global, in SPL 8-45
 - in SPL 8-19
 - local, in SPL 8-47
 - scope of SPL variable 8-43
 - unknown values in IF 8-61

with same name as a
keyword 8-22

View
creating synonym for 7-70
dropping 7-141
privileges 7-98
privileges with GRANT 7-183
restrictions with UNION
operator 7-287
updating 7-326
virtual column 7-99
with SELECT * notation 7-98
View Name segment 7-438
View, updatable 7-100

W

WAIT keyword, in the SET LOCK
MODE statement 7-311
WARNING keyword, in the
WHENEVER statement 7-337,
7-340
Warnings
with stored procedures 8-10
WEEKDAY function
syntax in expression 7-383
WHENEVER statement, syntax and
use 7-337
WHERE CURRENT OF keywords
syntax
in DELETE 7-122
in UPDATE 7-325
use
in UPDATE 7-333
WHERE keyword
joining tables 7-277
setting descriptions of
items 7-293
syntax
in DELETE 7-122
in SELECT 7-258
in UPDATE 7-325
use
in DELETE 7-123
in UPDATE 7-332
with a subquery 7-272
with ALL keyword 7-276
with ANY keyword 7-276

with BETWEEN keyword 7-272
with IN keyword 7-273
with IS keyword 7-273
with LIKE keyword 7-273
with MATCHES keyword 7-273
with relational operator 7-272
with SOME keyword 7-276

WHILE keyword
in CONTINUE statement 8-40
in EXIT 8-50
WHILE statement
looping in a stored
procedure 8-27
syntax 8-84
with NULL expressions 8-84
Wildcard characters, with LIKE or
MATCHES 7-427
WITH APPEND keywords, in the
SET DEBUG FILE TO
statement 7-291
WITH CHECK keywords
syntax in CREATE VIEW 7-97
use in CREATE VIEW 7-100
WITH GRANT keywords
syntax in GRANT 7-175
use in GRANT 7-181
WITH HOLD keywords
syntax in DECLARE 7-107
use in DECLARE 7-112, 7-120
WITH keyword, syntax in CREATE
DATABASE 7-49
WITH LISTING IN keywords
warnings in a stored
procedure 8-10
WITH LOG IN keywords, syntax in
START DATABASE 7-317
WITH MAX keywords
relationship with COUNT
field 7-294
WITH NO LOG keywords
syntax
in CREATE TABLE 7-75
in SELECT 7-285
use
in CREATE TABLE 7-91
in SELECT 7-287
WITH RESUME keywords, in
RETURN 8-76

WITHOUT HEADINGS keywords,
in the OUTPUT statement 7-216

X

X/Open compliance level Intro-22

Y

YEAR function
syntax in expression 7-383
use in expression 7-385
YEAR keyword
syntax
in DATETIME data type 7-368
in INTERVAL data type 7-414
use
as DATETIME field
qualifier 3-8, 7-417
as INTERVAL field
qualifier 3-14, 7-420

Symbols

. 6-13
||
concatenation operator 7-372