Informix® Guide to SQL

Tutorial

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

Informix Dynamic Server, Developer Edition, Version 7.3
Informix Dynamic Server, Workgroup Edition, Version 7.3

February 1998
Part No. 000-4366

Published by INFORMIX® Press

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

Documentation Team: Brian Deutscher, Evelyn Eldridge-Diaz, Geeta Karmarkar

Table of Contents

Introduction

About This Manual . . . 3
- Types of Users . . . 3
- Software Dependencies . . . 4
- Assumptions About Your Locale . . . 4
- Demonstration Databases . . . 5

New Features . . . 5
- New Features in Version 7.3 . . . 6
- New Features in Version 8.2 . . . 6

Documentation Conventions . . . 7
- Typographical Conventions . . . 8
- Icon Conventions . . . 9
- Sample-Code Conventions . . . 11

Additional Documentation . . . 12
- On-Line Manuals . . . 12
- Printed Manuals . . . 13
- Error Message Files . . . 13
- Documentation Notes, Release Notes, Machine Notes . . . 13
- Related Reading . . . 15

Compliance with Industry Standards . . . 15

Informix Welcomes Your Comments . . . 16

Section I Using Basic SQL

Chapter 1 Database Concepts

Illustration of a Data Model . . . 1-4
- Storing Data . . . 1-6
- Querying Data . . . 1-6
- Modifying Data . . . 1-9

Concurrent Use and Security 1-9
Controlling Database Use 1-9
Centralized Management 1-10
Important Database Terms 1-11
The Relational Model 1-11
Tables. 1-12
Columns. 1-13
Rows 1-13
Operations on Tables 1-13
Structured Query Language 1-15
Standard SQL 1-15
Informix SQL and ANSI SQL 1-16
ANSI-Compliant Databases 1-17
GLS Databases 1-17
Database Software 1-17
Applications 1-18
Database Server 1-18
Interactive SQL 1-18
General Programming 1-19
Summary 1-19

Chapter 2 **Composing Simple SELECT Statements**

Introducing the SELECT Statement 2-4
Some Basic Concepts 2-5
The Forms of SELECT 2-10
Special Data Types 2-11
Single-Table SELECT Statements 2-11
Selecting All Columns and Rows 2-12
Selecting Specific Columns 2-18
Using the WHERE Clause. 2-28
Creating a Comparison Condition 2-29
Using a FIRST Clause to Select Specific Rows 2-46
Expressions and Derived Values 2-49
Using Functions in SELECT Statements 2-57
Using Stored Procedures in SELECT Statements 2-86
Multiple-Table SELECT Statements 2-88
Creating a Cartesian Product. 2-88
Creating a Join. 2-90
Some Query Shortcuts 2-96
Selecting Tables from a Database Other Than the Current Database . 2-101
Summary 2-102

Chapter 3 **Composing Advanced SELECT Statements**

Using the GROUP BY and HAVING Clauses 3-4
- Using the GROUP BY Clause 3-4
- Using the HAVING Clause 3-8

Creating Advanced Joins 3-10
- Self-Joins 3-11
- Outer Joins 3-20

Subqueries in SELECT Statements 3-30
- Using ALL 3-31
- Using ANY 3-32
- Single-Valued Subqueries 3-33
- Correlated Subqueries 3-35
- Using EXISTS 3-36

Set Operations 3-39
- Union 3-40
- Intersection 3-48
- Difference. 3-50

Summary 3-52

Chapter 4 **Modifying Data**

Statements That Modify Data 4-3

Deleting Rows 4-4
- Deleting All Rows of a Table 4-4
- Deleting a Known Number of Rows. 4-5
- Deleting an Unknown Number of Rows 4-5
- Complicated Delete Conditions 4-6

Inserting Rows 4-7
- Single Rows 4-7
- Multiple Rows and Expressions 4-10
- Restrictions on the Insert Selection 4-11

Updating Rows 4-12
- Selecting Rows to Update 4-13
- Updating with Uniform Values 4-14
- Restrictions on Updates 4-15
- Updating with Selected Values 4-15
- Using a CASE Expression to Update a Column 4-16
- Using a Join to Update a Column. 4-17

Privileges on a Database 4-17
- Database-Level Privileges 4-18
- Table-Level Privileges 4-18
- Displaying Table Privileges. 4-19

Data Integrity 4-20
Entity Integrity 4-21
Semantic Integrity 4-21
Referential Integrity 4-22
Object Modes and Violation Detection 4-26
Interrupted Modifications 4-28
Transactions 4-30
Transaction Logging. 4-30
Specifying Transactions 4-32
Backups and Logs with Informix Database Servers 4-33
Concurrency and Locks 4-34
Data Replication 4-35
Informix Database Server Data Replication 4-36
Summary 4-37

Chapter 5 Programming with SQL

SQL in Programs 5-4
SQL in SQL APIs 5-4
SQL in Application Languages 5-5
Static Embedding 5-5
Dynamic Statements 5-5
Program Variables and Host Variables 5-6
Calling the Database Server 5-8
SQL Communications Area 5-8
SQLCODE Field 5-9
SQLERRD Array 5-10
SQLWARN Array 5-11
SQLERRM Character Array 5-13
SQLSTATE Value 5-13
Retrieving Single Rows 5-14
Data-Type Conversion 5-15
Working with Null Data 5-16
Dealing with Errors 5-17
Retrieving Multiple Rows 5-20
Declaring a Cursor 5-20
Opening a Cursor 5-21
Fetching Rows. 5-22
Cursor Input Modes. 5-23
Active Set of a Cursor 5-24
Using a Cursor: A Parts Explosion 5-27

Dynamic SQL 5-29
Preparing a Statement 5-30
Executing Prepared SQL. 5-31
Dynamic Host Variables 5-32
Freeing Prepared Statements 5-33
Quick Execution 5-33
Embedding Data-Definition Statements 5-34
Embedding Grant and Revoke Privileges 5-34
Summary 5-37

Chapter 6 Modifying Data Through SQL Programs

Using DELETE 6-3
Direct Deletions. 6-4
Deleting with a Cursor 6-7
Using INSERT 6-9
Using an Insert Cursor 6-9
Rows of Constants. 6-12
An Insert Example. 6-12
Using UPDATE 6-15
Using an Update Cursor. 6-15
Cleaning Up a Table 6-17
Summary 6-18

Chapter 7 Programming for a Multiuser Environment

Concurrency and Performance 7-3
Locking and Integrity 7-3
Locking and Performance 7-4
Concurrency Issues 7-4
How Locks Work 7-6
Kinds of Locks 7-7
Lock Scope 7-7
Duration of a Lock. 7-11
Locks While Modifying 7-11
Setting the Isolation Level 7-12
Comparing SET TRANSACTION with SET ISOLATION . . . 7-12
ANSI Read Uncommitted and Informix Dirty Read Isolation . . 7-14
ANSI Read Committed and Informix Committed Read Isolation. 7-14
Informix Cursor Stability Isolation 7-15
ANSI Serializable, ANSI Repeatable Read, and
Informix Repeatable Read Isolation. 7-17

Controlling Data Modification with Access Modes 7-18
Setting the Lock Mode 7-19
Waiting for Locks. 7-19
Not Waiting for Locks 7-19
Waiting a Limited Time 7-20
Handling a Deadlock 7-20
Handling External Deadlock 7-21
Simple Concurrency 7-21
Hold Cursors 7-21
Summary 7-23

Section II Using Advanced SQL

Chapter 8 Creating and Using Stored Procedures

Introduction to Stored Procedures and SPL 8-3
What You Can Do with Stored Procedures 8-4
Relationship Between SQL and a Stored Procedure 8-4
Stored-Procedure Behavior for Dynamic Server
with AD and XP Options 8-5
Creating and Using Stored Procedures 8-6
Creating a Procedure 8-6
Creating a Procedure in a Program. 8-7
Commenting and Documenting a Procedure 8-8
Diagnosing Compile-Time Errors 8-8
Looking at Compile-Time Warnings 8-10
Generating the Text or Documentation 8-11
Executing a Procedure 8-12
Executing a Stored Procedure Dynamically. 8-14
Debugging a Procedure 8-14
Privileges on Stored Procedures 8-17
Privileges Necessary at Creation 8-18
Privileges Necessary at Execution 8-18
Revoking Privileges 8-20
Variables and Expressions 8-20
SPL Variables 8-20
SPL Expressions 8-25
Program Flow Control 8-26
Branching 8-26
Looping 8-27
Function Handling 8-28

Passing Information to and from a Procedure 8-29
Returning Results 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
Summary 8-36

Chapter 9 Creating and Using Triggers

When to Use Triggers 9-3
How to Create a Trigger 9-4
Assigning a Trigger Name 9-5
Specifying the Trigger Event 9-5
Defining the Triggered Actions 9-6
A Complete CREATE TRIGGER Statement 9-7
Using Triggered Actions 9-7
Using BEFORE and AFTER Triggered Actions 9-7
Using FOR EACH ROW Triggered Actions 9-9
Using Stored Procedures as Triggered Actions 9-11
Reentrant Triggers for Dynamic Server 9-13
Tracing Triggered Actions 9-13
Example of TRACE Statements in a Stored Procedure 9-14
Example of TRACE Output. 9-14
Generating Error Messages 9-15
Applying a Fixed Error Message 9-15
Generating a Variable Error Message 9-17
Summary 9-18

Index

Introduction

About This Manual 3
Types of Users 3
Software Dependencies 4
Assumptions About Your Locale 4
Demonstration Databases 5

New Features 5
New Features in Version 7.3 6
New Features in Version 8.2 6

Documentation Conventions 7
Typographical Conventions 8
Icon Conventions 9
Comment Icons 9
Feature, Product, and Platform Icons 10
Compliance Icons 11
Sample-Code Conventions 11

Additional Documentation 12
On-Line Manuals 12
Printed Manuals 13
Error Message Files 13
Documentation Notes, Release Notes, Machine Notes 13
Related Reading 15

Compliance with Industry Standards 15

Informix Welcomes Your Comments 16

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

About This Manual

This manual shows how to use basic and advanced Structured Query Language (SQL) to access and manipulate the data in your databases. It discusses the data manipulation language (DML) statements as well as triggers and stored procedures, which DML statements often use.

This manual is one of a series of manuals that discusses the Informix implementation of Structured Query Language (SQL). This manual shows how to use basic and advanced SQL. The *Informix Guide to SQL: Syntax* contains all the syntax descriptions for SQL and stored procedure language (SPL). The *Informix Guide to SQL: Reference* provides reference information for aspects of SQL other than the language statements. The *Informix Guide to Database Design and Implementation* shows how to use SQL to implement and manage your databases.

Types of Users

This manual is for the following users:

- Database-application programmers
- Database users
- Database administrators

This manual assumes that you have the following background:

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

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

Software Dependencies

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

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

Assumptions About Your Locale

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

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

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

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

Demonstration Databases

The DB-Access utility, which is provided with your Informix database server products, includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. You can use SQL scripts provided with DB-Access to derive a second database, called **sales_demo**. This database illustrates a dimensional schema for data-warehousing applications. Sample command files are also included for creating and populating these databases.

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

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

New Features

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

New Features in Version 7.3

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

- Reliability, availability, and serviceability
- Performance
- Windows NT-specific features
- Application migration
- Manageability

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

This manual includes information about the following new features:

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

New Features in Version 8.2

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

- Global Language Support (GLS)
- New aggregates: STDEV, RANGE, and VARIANCE

- New TABLE lock mode for the LOCK MODE clause of ALTER TABLE and CREATE TABLE statement
- Support for specifying a lock on one or more rows for the Cursor Stability isolation level

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

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

Documentation Conventions

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

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Sample-code conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
`monospace`	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of feature-, product-, platform-, or compliance-specific information within a table or section.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

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

Icon Conventions

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

Comment Icons

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

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

Feature, Product, and Platform Icons

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

Icon	Description
AD/XP	Identifies information that is specific to Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options.
E/C	Identifies information that is specific to the INFORMIX-ESQL/C product.
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature.
IDS	Identifies information that is specific to Dynamic Server and its editions. However, in some cases, the identified section applies only to Informix Dynamic Server and not to Informix Dynamic Server, Workgroup and Developer Editions. Such information is clearly identified.
UNIX	Identifies information that is specific to UNIX platforms.
W/D	Identifies information that is specific to Informix Dynamic Server, Workgroup and Developer Editions.
WIN NT	Identifies information that is specific to the Windows NT environment.

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

Compliance Icons

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

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

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

Sample-Code Conventions

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

```
CONNECT TO stores7
...

DELETE FROM customer
    WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

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

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

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

Additional Documentation

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

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

On-Line Manuals

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

Printed Manuals

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

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

Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. For a detailed description of these error messages, refer to *Informix Error Messages* in Answers OnLine.

UNIX

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

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

♦

WIN NT

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

Documentation Notes, Release Notes, Machine Notes

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

UNIX

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

On-Line File	Purpose
SQLTDOC_*x.y*	The documentation-notes file for your version of this manual describes features that are not covered in the manual or that have been modified since publication. Replace *x.y* in the filename with the version number of your database server to derive the name of the documentation-notes file for this manual.
SERVERS_*x.y*	The release-notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds. Replace *x.y* in the filename with the version number of your database server to derive the name of the release-notes file.
IDS_*x.y*	The machine-notes file describes any special actions that are required to configure and use Informix products on your computer. Machine notes are named for the product described. Replace *x.y* in the filename with the version number of your database server to derive the name of the machine-notes file.

♦

WIN NT

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

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

Machine notes do not apply to Windows NT platforms. ♦

Related Reading

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

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

Compliance with Industry Standards

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

Informix Welcomes Your Comments

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

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

Write to us at the following address:

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

If you prefer to send email, our address is:

doc@informix.com

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

650-926-6571

We appreciate your feedback.

Using Basic SQL

Section I

Chapter

1

Database Concepts

Illustration of a Data Model 1-4
Storing Data 1-6
Querying Data 1-6
Modifying Data. 1-9

Concurrent Use and Security 1-9
Controlling Database Use 1-9
Centralized Management 1-10

Important Database Terms 1-11
The Relational Model. 1-11
Tables 1-12
Columns 1-13
Rows 1-13
Operations on Tables 1-13

Structured Query Language. 1-15
Standard SQL 1-15
Informix SQL and ANSI SQL 1-16
ANSI-Compliant Databases 1-17
GLS Databases 1-17

Database Software 1-17
Applications. 1-18
Database Server 1-18
Interactive SQL 1-18
General Programming 1-19

Summary 1-19

This book is about databases and how you can use Informix software to exploit them. As you start reading, keep in mind the following fundamental database characteristics:

- A database comprises not only data but also a plan, or *model,* of the data.
- A database can be a common resource, used concurrently by many people.

Your real use of a database begins with the SELECT statement, which is described in Chapter 2, "Composing Simple SELECT Statements." If you are in a hurry, and if you know at least a little about databases, turn to it now.

This chapter covers the fundamental concepts of databases and defines some terms that are used throughout the book. The chapter emphasizes the following topics:

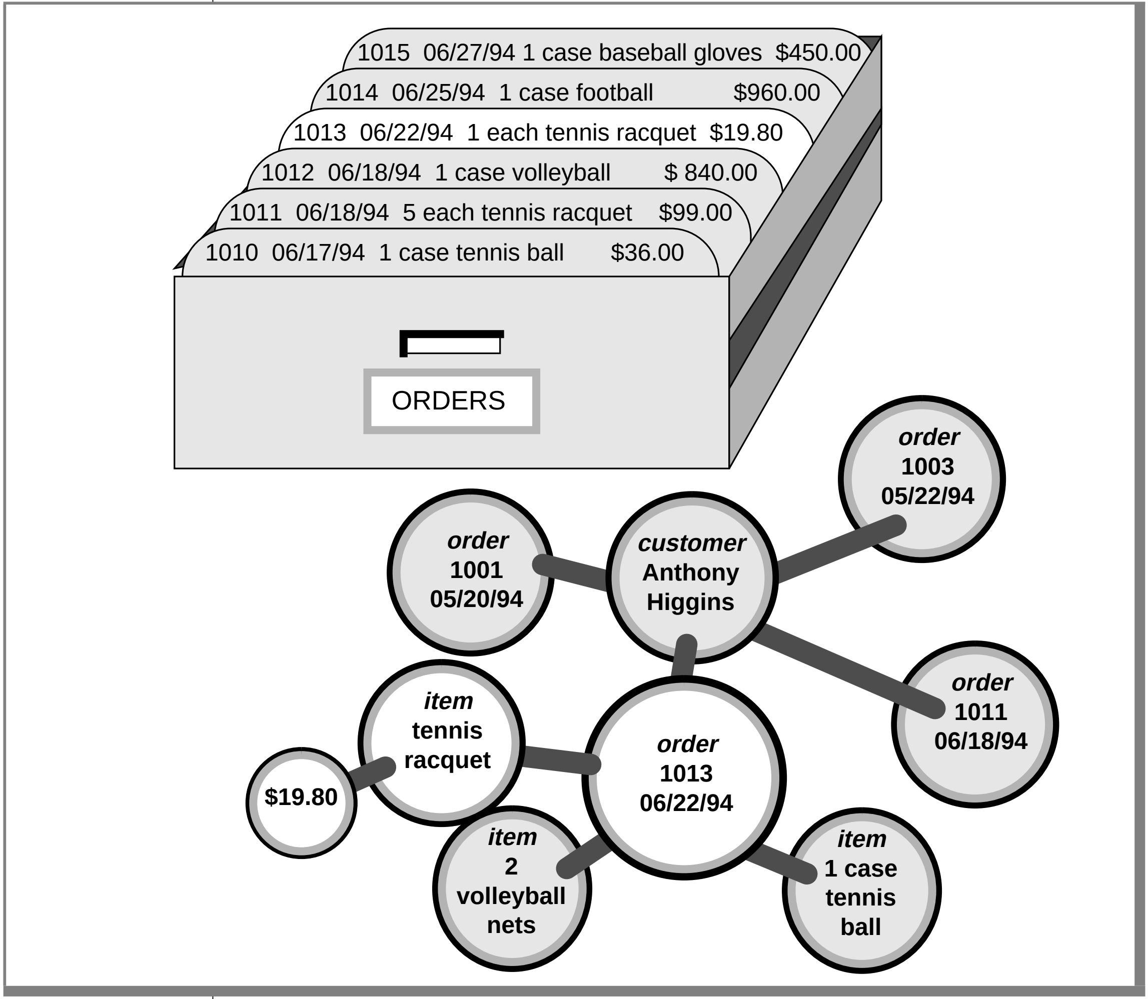
- How does the data model differentiate a database from a file?
- What issues are involved when many users use the database as a common resource?
- What terms are used to describe the main components of a database?
- What language is used to create, query, and modify a database?
- What are the main parts of the software that manages a database, and how do these parts work with each other?

Illustration of a Data Model

The principal difference between information collected in a database versus information collected in a file is the way the data is organized. A flat file is organized physically; certain items precede or follow other items. But the contents of a database are organized according to a *data model*. A data model is a plan, or map, that defines the units of data and specifies how each unit is related to the others.

For example, a number can appear in either a file or a database. In a file, it is simply a number that occurs at a certain point in the file. A number in a database, however, has a role that the data model assigns to it. It might be a *price* that is associated with a *product* that was sold as one *item* of an *order* that was placed by a *customer*. Each of these components, price, product, item, order, and customer, also has a role that the data model specifies. See Figure 1-1 for an illustration of a data model.

Figure 1-1
The Advantage of Using a Data Model


1015 06/27/94 1 case baseball gloves $450.00
1014 06/25/94 1 case football $960.00
1013 06/22/94 1 each tennis racquet $19.80
1012 06/18/94 1 case volleyball $ 840.00
1011 06/18/94 5 each tennis racquet $99.00
1010 06/17/94 1 case tennis ball $36.00
ORDERS
order 1003 05/22/94
order 1001 05/20/94
customer Anthony Higgins
order 1011 06/18/94
item tennis racquet
$19.80
order 1013 06/22/94
item 2 volleyball nets
item 1 case tennis ball

The data model is designed when the database is created. Units of data are then inserted according to the plan that the model lays out. Some books use the term *schema* instead of *data model*.

Storing Data

Another difference between a database and a file is that the organization of the database is stored with the database.

A file can have a complex inner structure, but the definition of that structure is not within the file; it is in the programs that create or use the file. For example, a document file that a word-processing program stores might contain very detailed structures describing the format of the document. However, only the word-processing program can decipher the contents of the file because the structure is defined within the program, not within the file.

A data model, however, is contained in the database it describes. It travels with the database and is available to any program that uses the database. The model defines not only the names of the data items but also their data types, so a program can adapt itself to the database. For example, a program can find out that, in the current database, a *price* item is a decimal number with eight digits, two to the right of the decimal point; then it can allocate storage for a number of that type. How programs work with databases is the subject of Chapter 5, "Programming with SQL," and Chapter 6, "Modifying Data Through SQL Programs."

Querying Data

Another difference between a database and a file is the way you can interrogate them. You can search a file sequentially, looking for particular values at particular physical locations in each line or record. That is, you might ask "What records have the number 1013 in the first field?" Figure 1-2 shows this type of search.

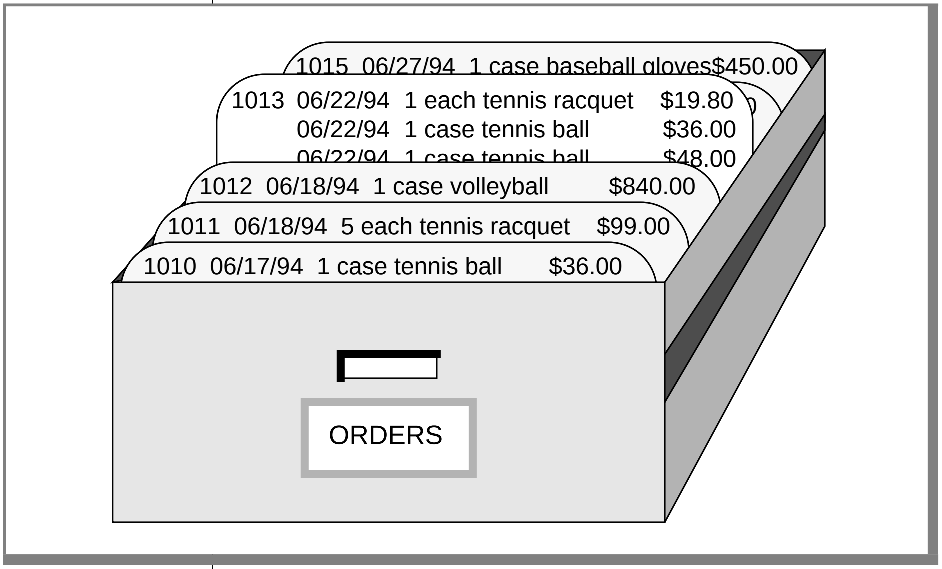
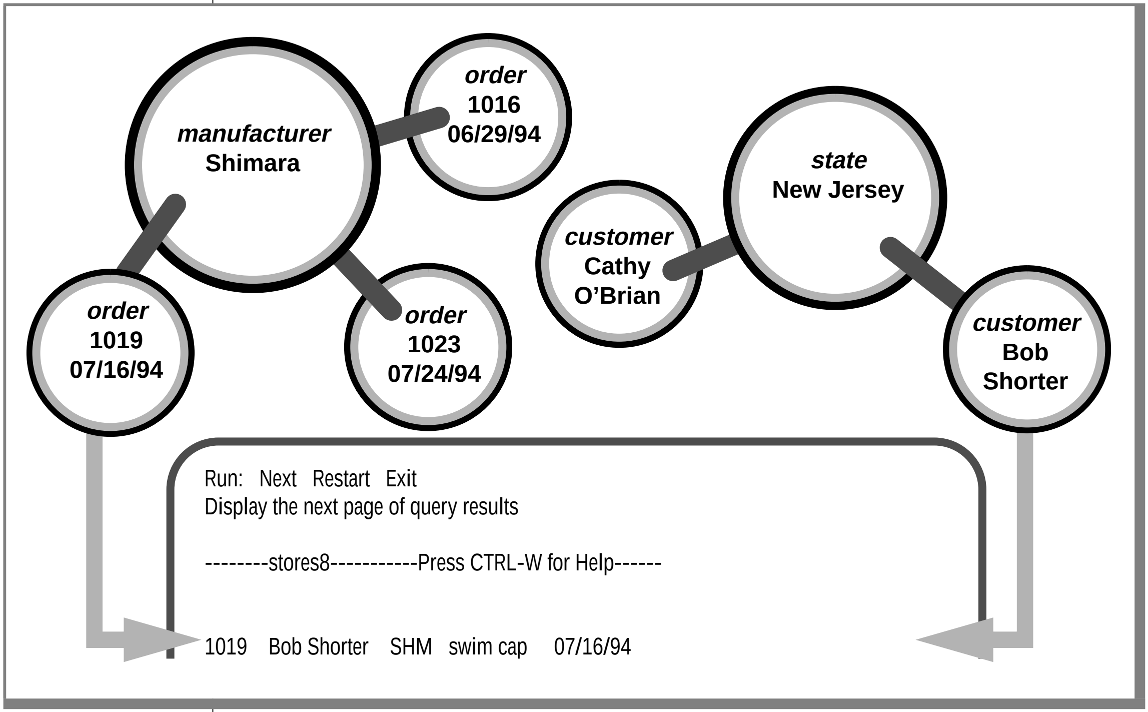

1015 06/27/94 1 case baseball gloves $450.00
1013 06/22/94 1 each tennis racquet $19.80
06/22/94 1 case tennis ball $36.00
06/22/94 1 case tennis ball $48.00
1012 06/18/94 1 case volleyball $840.00
1011 06/18/94 5 each tennis racquet $99.00
1010 06/17/94 1 case tennis ball $36.00
ORDERS

Figure 1-2
Searching a File Sequentially

In contrast, when you query a database, you use the terms that its model defines. You can query the database with questions such as, "What *orders* have been placed for *products* made by the Shimara Corporation, by *customers* in New Jersey, with *ship dates* in the third quarter?" Figure 1-3 shows this type of query.

Figure 1-3
Querying a Database



```
Run:   Next   Restart   Exit
Display the next page of query results

--------stores8-----------Press CTRL-W for Help------


1019    Bob Shorter    SHM   swim cap     07/16/94
```

In other words, when you interrogate data that is stored in a file, you must state your question in terms of the physical layout of the file. When you query a database, you can ignore the arcane details of computer storage and state your query in terms that reflect the real world, at least to the extent that the data model reflects the real world.

In this manual, Chapter 2, "Composing Simple SELECT Statements," and Chapter 3, "Composing Advanced SELECT Statements," discuss the language you use to make queries.

For information about how to build and implement your data model, see the *Informix Guide to Database Design and Implementation*.

Modifying Data

The model also makes it possible to modify the contents of the database with less chance for error. You can query the database with statements such as "Find every *stock item* with a *manufacturer* of Presta or Schraeder, and increase its *price* by 13 percent." You state changes in terms that reflect the meaning of the data. You do not have to waste time and effort thinking about details of fields within records in a file, so the chances for error are less.

The statements you use to modify stored data are covered in Chapter 4, "Modifying Data."

Concurrent Use and Security

A database can be a common resource for many users. Multiple users can query and modify a database simultaneously. The database server (the program that manages the contents of all databases) ensures that the queries and modifications are done in sequence and without conflict.

Having concurrent users on a database provides great advantages but also introduces new problems of security and privacy. Some databases are private; individuals set them up for their own use. Other databases contain confidential material that must be shared but among only a select group of persons; still other databases provide public access.

Controlling Database Use

Informix database software provides the means to control database use. When you design a database, you can perform any of the following functions:

- Keep the database completely private
- Open its entire contents to all users or to selected users
- Restrict the selection of data that some users can view. (In fact, you can reveal entirely different selections of data to different groups of users.)

- Allow specified users to view certain items but not modify them
- Allow specified users to add new data but not modify old data
- Allow specified users to modify all, or specified items of, existing data
- Ensure that added or modified data conforms to the data model

For information about how to grant and limit access to your database, see the *Informix Guide to Database Design and Implementation*.

Centralized Management

Databases that are used by many people are highly valuable and must be protected as important business assets. You create a significant problem when you compile a store of valuable data and simultaneously allow many employees to access it: protecting data while maintaining performance. The database server lets you centralize these tasks.

Databases must be guarded against loss or damage. The hazards are many: failures in software and hardware, and the risks of fire, flood, and other natural disasters. Losing an important database creates a huge potential for damage. The damage could include not only the expense and difficulty of re-creating the lost data but also the loss of productive time by the database users as well as the loss of business and good will while users cannot work. A plan for regular backups helps avoid or mitigate these potential disasters.

A large database that many people use must be maintained and tuned. Someone must monitor its use of system resources, chart its growth, anticipate bottlenecks, and plan for expansion. Users will report problems in the application programs; someone must diagnose these problems and correct them. If rapid response is important, someone must analyze the performance of the system and find the causes of slow responses.

Important Database Terms

You should know two sets of terms before you begin the next chapter. One set of terms describes the database and the data model; the other set describes the computer programs that manage the database. This section defines the terms that describe the database and the data model. For the terms that apply to programs that manage a database, see "Database Software" on page 1-17.

The Relational Model

Informix databases are *relational* databases. In technical terms, that means that the data model by which an Informix database is organized is based on the relational calculus devised by E. F. Codd. In practical terms, it means that all data is presented in the form of *tables* with *rows* and *columns*.

The relational model is a way of organizing data to reflect the world. It uses the following simple corresponding relationship.

Relationship	Description
table = entity	A table represents all that the database knows about one subject or kind of thing.
column = attribute	A column represents one feature, characteristic, or fact that is true of the table subject.
row = instance	A row represents one individual instance of the table subject.

Some rules apply about how you choose entities and attributes, but they are important only when you are designing a new database. (For complete information about database design, see the *Informix Guide to Database Design and Implementation*.) The data model in an existing database is already set. To use the database, you need to know only the names of the tables and columns and how they correspond to the real world.

Tables

A database is a collection of information that is grouped into one or more tables. A table is an array of data *items* organized into rows and columns. A demonstration database is distributed with every Informix product. A partial table from the demonstration database follows.

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
...	...	...	...	...	...
313	ANZ	swim cap	60.00	case	12/box

A table represents all that the database administrator (DBA) knows about one *entity*, one type of thing that the database describes. The example table, **stock**, represents all that the DBA knows about the merchandise that is stocked by a sporting-goods store. Other tables in the demonstration database represent such entities as **customer** and **orders**.

Think of a database as a collection of tables. To create a database is to create a set of tables. The right to query or modify tables can be controlled on a table-by-table basis, so that some users can view or modify some tables but not others.

Columns

Each column of a table stands for one *attribute*, which is one characteristic, feature, or fact that is true of the subject of the table. The **stock** table has columns for the following facts about items of merchandise: stock numbers, manufacturer codes, descriptions, prices, and units of measure.

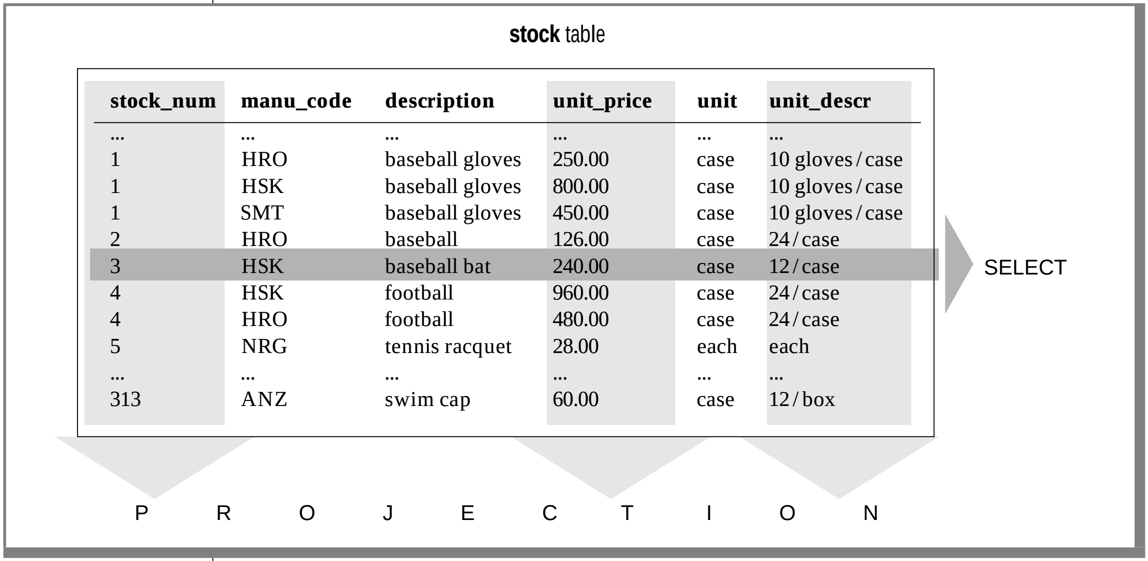
Rows

Each row of a table stands for one *instance* of the subject of the table, which is one particular example of that entity. Each row of the **stock** table stands for one item of merchandise that the sporting-goods store sells.

Operations on Tables

Because a database is really a collection of tables, database operations are operations on tables. The relational model supports three fundamental operations: selection, projection, and joining. Figure 1-4 shows the selection and projection operations. (All three operations are defined in more detail, with many examples, in Chapter 2, "Composing Simple SELECT Statements," and Chapter 3, "Composing Advanced SELECT Statements.")

Figure 1-4
Illustration of Selection and Projection



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
...	...	...	...	...	...
313	ANZ	swim cap	60.00	case	12/box

When you *select* data from a table, you are choosing certain rows and ignoring others. For example, you can query the **stock** table by asking the database management system to “select all rows in which the manufacturer code is HSK and the unit price is between 200.00 and 300.00.”

When you *project* from a table, you are choosing certain columns and ignoring others. For example, you can query the **stock** table by asking the database management system to “project the **stock_num**, **unit_descr**, and **unit_price** columns.”

A table contains information about only one entity; when you want information about multiple entities, you must *join* their tables. You can join tables in many ways. (The join operation is the subject of Chapter 3, “Composing Advanced SELECT Statements.”)

Structured Query Language

Most computer software has not yet reached a point where you can literally ask a database, “What orders have been placed by customers in New Jersey with ship dates in the third quarter?” You must still phrase questions in a restricted syntax that the software can easily parse. You can pose the same question to the demonstration database in the following terms:

```
SELECT * FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
        AND customer.state = 'NJ'
        AND orders.ship_date
        BETWEEN DATE('7/1/96') AND DATE('9/30/96')
```

This question is a sample of Structured Query Language (SQL). It is the language that you use to direct all operations on the database. SQL is composed of statements, each of which begins with one or two keywords that specify a function. The Informix implementation of SQL includes a large number of SQL statements, from ALLOCATE DESCRIPTOR to WHENEVER.

All the SQL statements are specified in detail in the *Informix Guide to SQL: Syntax*. Most of the statements are used infrequently, when you set up or tune a database. People generally use three or four statements to query or update databases.

One statement, SELECT, is in almost constant use. SELECT is the only statement that you can use to retrieve data from the database. It is also the most complicated statement, and the next two chapters of this book explore its many uses.

Standard SQL

SQL and the relational model were invented and developed at IBM in the early and middle 1970s. Once IBM proved that it was possible to implement practical relational databases and that SQL was a usable language for manipulating them, other vendors began to provide similar products for non-IBM computers.

For reasons of performance or competitive advantage, or to take advantage of local hardware or software features, each SQL implementation differed in small ways from the others and from the IBM version of the language. To ensure that the differences remained small, a standards committee was formed in the early 1980s.

Committee X3H2, sponsored by the American National Standards Institute (ANSI), issued the SQL1 standard in 1986. This standard defines a core set of SQL features and the syntax of statements such as SELECT.

Informix SQL and ANSI SQL

The SQL version that Informix products support is compatible with standard SQL (it is also compatible with the IBM version of the language). However, it does contain *extensions* to the standard; that is, extra options or features for certain statements, and looser rules for others. Most of the differences occur in the statements that are not in everyday use. For example, few differences occur in the SELECT statement, which accounts for 90 percent of the SQL use for a typical person.

However, the extensions do exist and create a conflict. Thousands of Informix customers have embedded Informix-style SQL in programs and stored procedures. They rely on Informix to keep its language the same. Other customers require the ability to use databases in a way that conforms exactly to the ANSI standard. They rely on Informix to change its language to conform.

Informix resolves the conflict with the following compromise:

- The Informix version of SQL, with its extensions to the standard, is available by default.
- You can ask any Informix SQL language processor to check your use of SQL and post a warning flag whenever you use an Informix extension.

Wherever a difference exists between Informix and ANSI SQL, the *Informix Guide to SQL: Syntax* describes both versions. Because you probably intend to use only one version, simply ignore the version you do not need.

ANSI-Compliant Databases

Use the MODE ANSI keywords when you create a database to designate it as ANSI compliant. Within such a database, certain characteristics of the ANSI standard apply. For example, all actions that modify data take place within a transaction automatically, which means that the changes are made in their entirety or not at all. Differences in the behavior of ANSI-compliant databases are noted where appropriate in the statement descriptions in the *Informix Guide to SQL: Syntax*. For a detailed discussion of ANSI-compliant databases, see the *Informix Guide to Database Design and Implementation*.

GLS

GLS Databases

Informix database server products provide the Global Language Support (GLS) feature. In addition to U.S. ASCII English, GLS allows you to work in other locales and use non-ASCII characters in SQL data and identifiers. You can use the GLS feature to conform to the customs of a specific locale. The locale files contain culture-specific information such as various money and date formats and collation orders. For complete GLS information, see the *Informix Guide to GLS Functionality*.

Database Software

You access your database through two layers of sophisticated software. The top layer, or *application*, sends commands or queries to the database server. The application calls on the bottom layer, or *database server*, and gets back information. You command both layers when you use SQL.

Every program that uses data from a database operates in the same way; you use an application and database server in every case.The application interacts with the user, prepares and formats data, and sets up SQL statements. The database server manages the database and interprets the SQL statements. All the applications make requests of the database server, and only the database server manipulates the database files on disk.

Applications

A *database application*, or simply *application*, is a program that uses the database. It does so by communicating with the database server. At its simplest, the application sends SQL statements to the database server, and the database server sends rows of data back to the application. Then the application displays the rows to you, its user.

Alternatively, you command the application to add new data to the database. It incorporates the new data as part of an SQL statement to insert a row and passes this statement to the database server for execution.

Several types of applications exist. Some allow you to access the database interactively with SQL; others present the stored data in different forms related to its use.

Database Server

The *database server* is the program that manages the contents of the database as they are stored on disk. The database server knows how tables, rows, and columns are actually organized in physical computer storage. The database server also interprets and executes all SQL commands.

Interactive SQL

To carry out the examples in this book, and to experiment with SQL and database design for yourself, you need a program that lets you execute SQL statements interactively. DB-Access and the Relational Object Manager are examples of such programs. They help you to compose SQL statements; then they pass your SQL statements to the database server for execution and display the results to you.

AD/XP

Dynamic Server with AD and XP Options does not support the Relational Object Manager program. ♦

General Programming

You can write programs that incorporate SQL statements and exchange data with the database server. That is, you can write a program to retrieve data from the database and format it however you choose. You can also write programs that take data from any source in any format, prepare it, and insert it into the database.

You can also write programs called stored procedures to work with database data and objects. The stored procedures that you write are stored directly in a database in tables. You can then execute a stored procedure from DB-Access, ROM, or an SQL application programming interface (SQL API) such as INFORMIX-ESQL/C.

Chapter 5, "Programming with SQL," and Chapter 6, "Modifying Data Through SQL Programs," present an overview of how SQL is used in programs.

Summary

A database contains a collection of related information but differs in a fundamental way from other methods of storing data. The database contains not only the data but also a data model that defines each data item and specifies its meaning with respect to the other items and to the real world.

More than one user can access and modify a database at the same time. Each user has a different view of the contents of a database, and each user's access to those contents can be restricted in several ways.

A relational database consists of tables, and the tables consist of columns and rows. The relational model supports three fundamental operations on tables: selections, projections, and joins.

To manipulate and query a database use SQL. IBM pioneered SQL and ANSI standardized it. Informix added extensions to the ANSI-defined language that you can use to your advantage. Informix tools also make it possible to maintain strict compliance with ANSI standards.

Two layers of software mediate all your work with databases. The bottom layer is always a database server that executes SQL statements and manages the data on disk and in computer memory. The top layer is one of many applications, some from Informix and some written by you or written by other vendors or your colleagues.

Chapter

2

Composing Simple SELECT Statements

Introducing the SELECT Statement 2-4
- Some Basic Concepts 2-5
  - Privileges 2-5
  - Relational Operations 2-5
  - Selection and Projection 2-6
  - Joining 2-8
- The Forms of SELECT 2-10
- Special Data Types. 2-11

Single-Table SELECT Statements 2-11
- Selecting All Columns and Rows. 2-12
  - Using the Asterisk Symbol (*) 2-12
  - Reordering the Columns 2-13
  - Sorting the Rows 2-13
- Selecting Specific Columns 2-18
  - ORDER BY and Non-English Data 2-25
  - Selecting Substrings 2-27
- Using the WHERE Clause 2-28
- Creating a Comparison Condition 2-29
  - Using Variable-Text Searches 2-37
  - Using Exact-Text Comparisons 2-38
  - Using a Single-Character Wildcard. 2-39
  - MATCHES and Non-English Data 2-42
  - Comparing for Special Characters 2-44
- Using a FIRST Clause to Select Specific Rows 2-46
  - FIRST Clause Without an ORDER BY Clause 2-46
  - FIRST Clause with an ORDER BY Clause 2-47
  - FIRST Clause in a Union Query 2-48
- Expressions and Derived Values 2-49
  - Arithmetic Expressions. 2-49
  - CASE Expressions 2-54
  - Sorting on Derived Columns 2-56

Using Functions in SELECT Statements 2-57
Aggregate Functions. 2-57
Time Functions 2-63
Date-Conversion Functions 2-68
String Manipulation Functions 2-71
Other Functions 2-79
Using Stored Procedures in SELECT Statements 2-86

Multiple-Table SELECT Statements 2-88
Creating a Cartesian Product 2-88
Creating a Join 2-90
Equi-Join. 2-90
Natural Join. 2-93
Multiple-Table Join 2-95
Some Query Shortcuts 2-96
Using Aliases 2-96
The INTO TEMP Clause 2-100

Selecting Tables from a Database Other Than the Current Database . . 2-101

Summary 2-102

SELECT is the most important and the most complex SQL statement. You can use it, along with the SQL statements INSERT, UPDATE, and DELETE, to manipulate data. You can use the SELECT statement in the following ways:

- By itself to retrieve data from a database
- As part of an INSERT statement to produce new rows
- As part of an UPDATE statement to update information

The SELECT statement is the primary way to query information in a database. It is your key to retrieving data in a program, report, screen form, or spreadsheet.

This chapter shows how you can use the SELECT statement to query on and retrieve data in a variety of ways from a relational database. It discusses how to tailor your statements to select columns or rows of information from one or more tables, how to include expressions and functions in SELECT statements, and how to create various join conditions between relational database tables.

This chapter introduces the basic methods for retrieving data from a relational database. More complex uses of SELECT statements, such as subqueries, outer joins, and unions, are discussed in Chapter 3, "Composing Advanced SELECT Statements." The syntax and usage for the SELECT statement are described in detail in the *Informix Guide to SQL: Syntax*.

Most examples in this chapter come from the tables in the demonstration database, which is installed with the software for your Informix SQL API or database utility. In the interest of brevity, the examples show only part of the data that is retrieved for each SELECT statement. For information on the structure and contents of the demonstration database, see the *Informix Guide to SQL: Reference*. For emphasis, keywords are shown in uppercase letters in the examples, although SQL is not case sensitive.

Introducing the SELECT Statement

The SELECT statement is constructed of clauses that let you look at data in a relational database. These clauses let you select columns and rows from one or more database tables or views, specify one or more conditions, order and summarize the data, and put the selected data in a temporary table.

This chapter shows how to use five SELECT statement clauses. You must include these clauses in a SELECT statement in the following order:

1. SELECT clause
2. FROM clause
3. WHERE clause
4. ORDER BY clause
5. INTO TEMP clause

Only the SELECT and FROM clauses are required. These two clauses form the basis for every database query because they specify the tables and columns to be retrieved. Use one or more of the other clauses from the following list:

- Add a WHERE clause to select specific rows or create a *join* condition.
- Add an ORDER BY clause to change the order in which data is produced.
- Add an INTO TEMP clause to save the results as a table for further queries.

Two additional SELECT statement clauses, GROUP BY and HAVING, let you perform more complex data retrieval. They are introduced in Chapter 3, "Composing Advanced SELECT Statements." Another clause, INTO, specifies the program or host variable to receive data from a SELECT statement in INFORMIX-NewEra and SQL APIs. Complete syntax and rules for using the SELECT statement are shown in the *Informix Guide to SQL: Syntax*.

Some Basic Concepts

The SELECT statement, unlike INSERT, UPDATE, and DELETE statements, does not modify the data in a database. It simply queries the data. Whereas only one user at a time can modify data, multiple users can query or *select* the data concurrently. The statements that modify data appear in Chapter 4, "Modifying Data." The syntax descriptions of the INSERT, UPDATE, and DELETE statements appear in the *Informix Guide to SQL: Syntax*.

In a relational database, a *column* is a data element that contains a particular type of information that occurs in every row in the table. A *row* is a group of related items of information about a single entity across all columns in a database table.

You can select columns and rows from a database table; from a *system-catalog table*, a special table that contains information on the database; or from a *view*, a virtual table created to contain a customized set of data. System catalog tables are described in the *Informix Guide to SQL: Reference*. Views are discussed in the *Informix Guide to Database Design and Implementation*.

Privileges

Before you make a query against data, make sure you have the Connect privilege on the database and the Select privilege on the table. These privileges are normally granted to all users. Database access privileges are discussed in the *Informix Guide to Database Design and Implementation* and in the GRANT and REVOKE statements in the *Informix Guide to SQL: Syntax*.

Relational Operations

A *relational operation* involves manipulating one or more tables, or *relations*, to result in another table. The three kinds of relational operations are selection, projection, and join. This chapter includes examples of selection, projection, and simple joining.

Selection and Projection

In relational terminology, *selection* is defined as taking the *horizontal* subset of rows of a single table that satisfies a particular condition. This kind of SELECT statement returns some of the rows and all of the columns in a table. Selection is implemented through the WHERE clause of a SELECT statement, as Query 2-1 shows.

Query 2-1

```
SELECT * FROM customer
    WHERE state = 'NJ'
```

Query Result 2-1 contains the same number of columns as the **customer** table, but only a subset of its rows. Because the data in the selected columns does not fit on one line of the DB-Access or the Relational Object Manager screen, the data is displayed vertically instead of horizontally.

Query Result 2-1

```
customer_num  119
fname    Bob
lname    Shorter
company  The Triathletes Club
address1 2405 Kings Highway
address2
city     Cherry Hill
state    NJ
zipcode  08002
phone    609-663-6079


customer_num  122
fname    Cathy
lname    O‘Brian
company  The Sporting Life
address1 543d Nassau
address2
city     Princeton
state    NJ
zipcode  08540
phone    609-342-0054
```

In relational terminology, *projection* is defined as taking a *vertical* subset from the columns of a single table that retains the unique rows. This kind of SELECT statement returns some of the columns and all of the rows in a table.

Projection is implemented through the select list in the SELECT clause of a SELECT statement, as Query 2-2 shows.

Query 2-2

```
SELECT UNIQUE city, state, zipcode
     FROM customer
```

Query Result 2-2 contains the same number of rows as the **customer** table, but it *projects* only a subset of the columns in the table.

Query Result 2-2

```
city            state zipcode

Bartlesville    OK    74006
Blue Island     NY    60406
Brighton        MA    02135
Cherry Hill     NJ    08002
Denver          CO    80219
Jacksonville    FL    32256
Los Altos       CA    94022
Menlo Park      CA    94025
Mountain View   CA    94040
Mountain View   CA    94063
Oakland         CA    94609
Palo Alto       CA    94303
Palo Alto       CA    94304
Phoenix         AZ    85008
Phoenix         AZ    85016
Princeton       NJ    08540
Redwood City    CA    94026
Redwood City    CA    94062
Redwood City    CA    94063
San Francisco   CA    94117
Sunnyvale       CA    94085
Sunnyvale       CA    94086
Wilmington      DE    19898
```

The most common kind of SELECT statement uses both selection and projection. A query of this kind, shown in Query 2-3, returns some of the rows and some of the columns in a table.

Query 2-3

```
SELECT UNIQUE city, state, zipcode
    FROM customer
    WHERE state = 'NJ'
```

Query Result 2-3 contains a subset of the rows and a subset of the columns in the **customer** table.

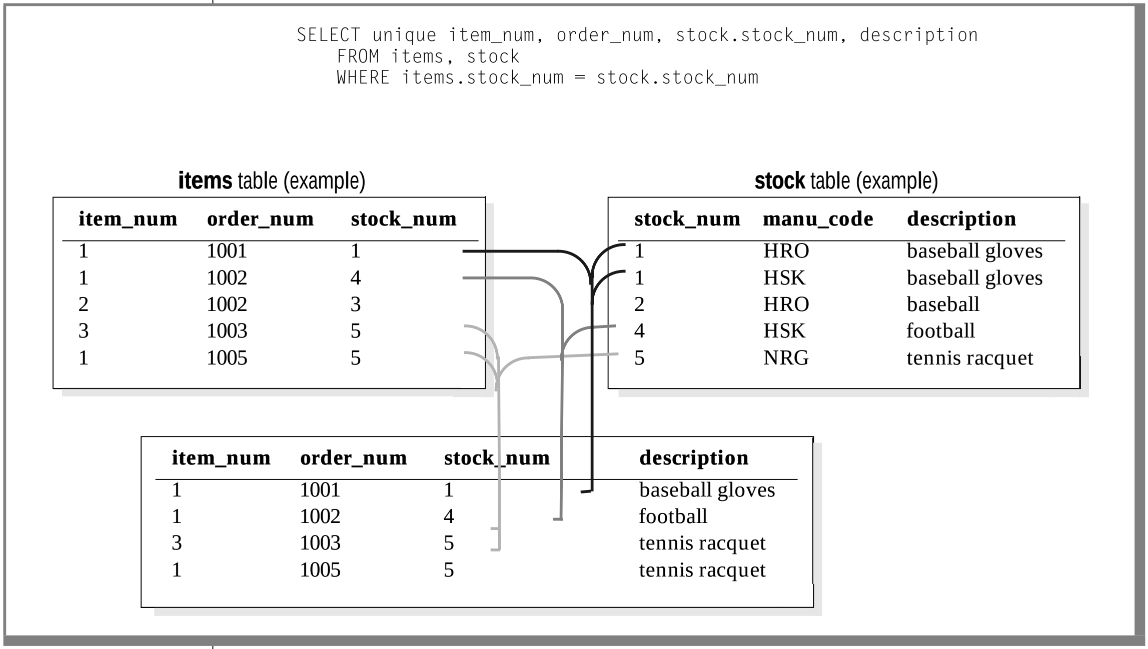
Query Result 2-3

```
city           state zipcode

Cherry Hill    NJ    08002
Princeton      NJ    08540
```

Joining

A join occurs when two or more tables are connected by one or more columns in common, creating a new table of results. The query in the example uses a subset of the **items** and **stock** tables to illustrate the concept of a join, as Figure 2-1 shows.

Figure 2-1
A Join Between Two Tables



```
SELECT unique item_num, order_num, stock.stock_num, description
    FROM items, stock
    WHERE items.stock_num = stock.stock_num
```

items table (example)

item_num	order_num	stock_num
1	1001	1
1	1002	4
2	1002	3
3	1003	5
1	1005	5

stock table (example)

stock_num	manu_code	description
1	HRO	baseball gloves
1	HSK	baseball gloves
2	HRO	baseball
4	HSK	football
5	NRG	tennis racquet

item_num	order_num	stock_num	description
1	1001	1	baseball gloves
1	1002	4	football
3	1003	5	tennis racquet
1	1005	5	tennis racquet

Query 2-4 joins the **customer** and **state** tables.

Query 2-4

```
SELECT UNIQUE city, state, zipcode, sname
    FROM customer, state
    WHERE customer.state = state.code
```

Query Result 2-4 consists of specified rows and columns from both the **customer** and **state** tables.

Query Result 2-4

```
city            state zipcode sname

Bartlesville    OK    74006   Oklahoma
Blue Island     NY    60406   New York
Brighton        MA    02135   Massachusetts
Cherry Hill     NJ    08002   New Jersey
Denver          CO    80219   Colorado
Jacksonville    FL    32256   Florida
Los Altos       CA    94022   California
Menlo Park      CA    94025   California
Mountain View   CA    94040   California
Mountain View   CA    94063   California
Oakland         CA    94609   California
Palo Alto       CA    94303   California
Palo Alto       CA    94304   California
Phoenix         AZ    85008   Arizona
Phoenix         AZ    85016   Arizona
Princeton       NJ    08540   New Jersey
Redwood City    CA    94026   California
Redwood City    CA    94062   California
Redwood City    CA    94063   California
San Francisco   CA    94117   California
Sunnyvale       CA    94085   California
Sunnyvale       CA    94086   California
Wilmington      DE    19898   Delaware
```

The Forms of SELECT

Although the syntax remains the same across all Informix products, the form of a SELECT statement and the location and formatting of the resulting output depends on the application. The examples in this chapter and in Chapter 3, "Composing Advanced SELECT Statements," display the SELECT statements and their output as they appear when you use the interactive Query-language option in DB-Access or the Relational Object Manager. You also can embed SELECT statements in a language such as INFORMIX-ESQL/C (where they are treated as executable code).

Special Data Types

With DB-Access, when you issue a SELECT statement that includes a VARCHAR, TEXT, or BYTE data type, the results of the query are displayed differently.

- If you execute a query on a VARCHAR column, the entire VARCHAR value is displayed, just as CHARACTER values are displayed.
- If you select a TEXT column, the contents of the TEXT column are displayed, and you can scroll through them.
- If you query on a BYTE column, the words `<BYTE value>` are displayed instead of the actual value.

Differences specific to VARCHAR, TEXT, and BYTE are noted as appropriate throughout this chapter.

GLS

You can issue a SELECT statement that queries NCHAR columns instead of CHAR columns or NVARCHAR columns instead of VARCHAR columns.

For complete GLS information, see the *Informix Guide to GLS Functionality*. For additional information on GLS and other data types, see the *Informix Guide to Database Design and Implementation* and the *Informix Guide to SQL: Reference*. ♦

Single-Table SELECT Statements

You can query a single table in a database in many ways. You can tailor a SELECT statement to perform the following actions:

- Retrieve all or specific columns
- Retrieve all or specific rows
- Perform computations or other functions on the retrieved data
- Order the data in various ways

Selecting All Columns and Rows

The most basic SELECT statement contains only the two required clauses, SELECT and FROM.

Using the Asterisk Symbol (*)

Query 2-5a specifies all the columns in the **manufact** table in a *select list*. A select list is a list of the column names or expressions that you want to project from a table.

Query 2-5a

```
SELECT manu_code, manu_name, lead_time
    FROM manufact
```

Query 2-5b uses the *wildcard* asterisk symbol (*), which is shorthand for the select list. The * represents the names of all the columns in the table. You can use the asterisk symbol (*) when you want all the columns, in their defined order.

Query 2-5b

```
SELECT * FROM manufact
```

Query 2-5a and Query 2-5b are equivalent and display the same results; that is, a list of every column and row in the **manufact** table. Query Result 2-5 shows the results as they would appear on a DB-Access or Relational Object Manager screen.

Query Result 2-5

```
manu_code manu_name       lead_time

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

Reordering the Columns

Query 2-6 shows how you can change the order in which the columns are listed by changing their order in your select list.

Query 2-6

```
SELECT manu_name, manu_code, lead_time
    FROM manufact
```

Query Result 2-6 includes the same columns as the previous query result, but because the columns are specified in a different order, the display is also different.

Query Result 2-6

```
manu_name       manu_code lead_time

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

Sorting the Rows

You can add an ORDER BY clause to your SELECT statement to direct the system to sort the data in a specific order.You must include the columns that you want to use in the ORDER BY clause in the select list either explicitly or implicitly.

An *explicit* select list, shown in Query 2-7a, includes all the column names that you want to retrieve.

Query 2-7a

```
SELECT manu_code, manu_name, lead_time
    FROM manufact
    ORDER BY lead_time
```

An *implicit* select list uses the asterisk symbol (*), as Query 2-7b shows.

Query 2-7b

```
SELECT * FROM manufact
    ORDER BY lead_time
```

Query 2-7a and Query 2-7b produce the same display. Query Result 2-7 shows a list of every column and row in the **manufact** table, in order of **lead_time**.

Query Result 2-7

```
manu_code manu_name       lead_time

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

Ascending Order

The retrieved data is sorted and displayed, by default, in *ascending* order. Ascending order is uppercase A to lowercase z for character data types, and lowest to highest value for numeric data types. DATE and DATETIME data is sorted from earliest to latest, and INTERVAL data is ordered from shortest to longest span of time.

Descending Order

Descending order is the opposite of ascending order, from lowercase z to uppercase A for character types and highest to lowest for numeric data types. DATE and DATETIME data is sorted from latest to earliest, and INTERVAL data is ordered from longest to shortest span of time. Query 2-8 shows an example of descending order.

Query 2-8

```
SELECT * FROM manufact
    ORDER BY lead_time DESC
```

The keyword DESC following a column name causes the retrieved data to be sorted in *descending* order, as Query Result 2-8 shows.

Query Result 2-8

```
manu_code manu_name       lead_time

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

You can specify any column (except TEXT or BYTE) in the ORDER BY clause, and the database server sorts the data based on the values in that column.

Sorting on Multiple Columns

You can also ORDER BY two or more columns, creating a *nested sort*. The default is still ascending, and the column that is listed first in the ORDER BY clause takes precedence.

Query 2-9 and Query 2-10 and corresponding query results show nested sorts. To modify the order in which selected data is displayed, change the order of the two columns that are named in the ORDER BY clause.

Query 2-9

```
SELECT * FROM stock
    ORDER BY manu_code, unit_price
```

In Query Result 2-9, the **manu_code** column data appears in alphabetical order and, within each set of rows with the same **manu_code** (for example, ANZ, HRO), the **unit_price** is listed in ascending order.

Query Result 2-9

```
stock_num manu_code description     unit_price unit unit_descr

        5 ANZ       tennis racquet      $19.80 each each
        9 ANZ       volleyball net      $20.00 each each
        6 ANZ       tennis ball         $48.00 case 24 cans/case
      313 ANZ       swim cap            $60.00 box  12/box
      201 ANZ       golf shoes          $75.00 each each
      310 ANZ       kick board          $84.00 case 12/case
      301 ANZ       running shoes       $95.00 each each
      304 ANZ       watch              $170.00 box  10/box
      110 ANZ       helmet             $244.00 case 4/case
      205 ANZ       3 golf balls       $312.00 case 24/case
        8 ANZ       volleyball         $840.00 case 24/case
      302 HRO       ice pack             $4.50 each each
      309 HRO       ear drops           $40.00 case 20/case
      .
      .
      .
      113 SHM       18-spd, assmbld    $685.90 each each
        5 SMT       tennis racquet      $25.00 each each
        6 SMT       tennis ball         $36.00 case 24 cans/case
        1 SMT       baseball gloves    $450.00 case 10 gloves/case
```

Query 2-10 shows the reversed order of the columns in the ORDER BY clause.

Query 2-10

```
    SELECT * FROM stock
         ORDER BY unit_price, manu_code
```

In Query Result 2-10, the data appears in ascending order of **unit_price** and, where two or more rows have the same **unit_price** (for example, $20.00, $48.00, $312.00), the **manu_code** is in alphabetical order.

Query Result 2-10

```
stock_num manu_code description     unit_price unit unit_descr

      302 HRO       ice pack            $4.50 each each
      302 KAR       ice pack            $5.00 each each
        5 ANZ       tennis racquet     $19.80 each each
        9 ANZ       volleyball net     $20.00 each each
      103 PRC       frnt derailleur    $20.00 each each
      106 PRC       bicycle stem       $23.00 each each
        5 SMT       tennis racquet     $25.00 each each
      .
      .
      .
      301 HRO       running shoes      $42.50 each each
      204 KAR       putter             $45.00 each each
      108 SHM       crankset           $45.00 each each
        6 ANZ       tennis ball        $48.00 case 24 cans/case
      305 HRO       first-aid kit      $48.00 case 4/case
      303 PRC       socks              $48.00 box  24 pairs/box
      311 SHM       water gloves       $48.00 box  4 pairs/box
      .
      .
      .
      110 HSK       helmet            $308.00 case 4/case
      205 ANZ       3 golf balls      $312.00 case 24/case
      205 HRO       3 golf balls      $312.00 case 24/case
      205 NKL       3 golf balls      $312.00 case 24/case
        1 SMT       baseball gloves   $450.00 case 10 gloves/case
        4 HRO       football          $480.00 case 24/case
      102 PRC       bicycle brakes    $480.00 case 4 sets/case
      111 SHM       10-spd, assmbld   $499.99 each each
      112 SHM       12-spd, assmbld   $549.00 each each
        7 HRO       basketball        $600.00 case 24/case
      203 NKL       irons/wedge       $670.00 case 2 sets/case
      113 SHM       18-spd, assmbld   $685.90 each each
        1 HSK       baseball gloves   $800.00 case 10 gloves/case
        8 ANZ       volleyball        $840.00 case 24/case
        4 HSK       football          $960.00 case 24/case
```

The order of the columns in the ORDER BY clause is important, and so is the position of the DESC keyword. Although the statements in Query 2-11 contain the same components in the ORDER BY clause, each produces a different result (not shown).

Query 2-11

```
SELECT * FROM stock
    ORDER BY manu_code, unit_price DESC

SELECT * FROM stock
    ORDER BY unit_price, manu_code DESC

SELECT * FROM stock
    ORDER BY manu_code DESC, unit_price

SELECT * FROM stock
    ORDER BY unit_price DESC, manu_code
```

Selecting Specific Columns

The previous section showed how to select and order all data from a table. However, often all you want to see is the data in one or more specific columns. Again, the formula is to use the SELECT and FROM clauses, specify the columns and table, and perhaps order the data in ascending or descending order with an ORDER BY clause.

If you want to find all the customer numbers in the **orders** table, use a statement such as the one in Query 2-12.

Query 2-12

```
SELECT customer_num FROM orders
```

Query Result 2-12 shows how the statement simply selects all data in the **customer_num** column in the **orders** table and lists the customer numbers on all the orders, including duplicates.

Query Result 2-12

```
customer_num

         104
         101
         104
         106
         116
         112
         117
         110
         111
         115
         104
         117
         104
         106
         110
         119
         120
         121
         122
         123
         124
         126
         127
```

The output includes several duplicates because some customers have placed more than one order. Sometimes you want to see duplicate rows in a projection. At other times, you want to see only the distinct values, not how often each value appears.

To suppress duplicate rows, include the keyword DISTINCT or its synonym UNIQUE at the start of the select list, as Query 2-13 shows.

Query 2-13

```
SELECT DISTINCT customer_num FROM orders

SELECT UNIQUE customer_num FROM orders
```

To produce a more readable list, Query 2-13 limits the display to show each customer number in the **orders** table only once, as Query Result 2-13 shows.

Query Result 2-13

```
customer_num

         101
         104
         106
         110
         111
         112
         115
         116
         117
         119
         120
         121
         122
         123
         124
         126
         127
```

Suppose you are handling a customer call, and you want to locate purchase order number DM354331. To list all the purchase order numbers in the **orders** table, use a statement such as the one that Query 2-14 shows.

Query 2-14

```
SELECT po_num FROM orders
```

Query Result 2-14 shows how the statement retrieves data in the **po_num** column in the **orders** table.

Query Result 2-14

```
po_num

 B77836
 9270
 B77890
 8006
 2865
 Q13557
 278693
 LZ230
 4745
 429Q
 B77897
 278701
 B77930
 8052
 MA003
 PC6782
 DM354331
 S22942
 Z55709
 W2286
 C3288
 W9925
 KF2961
```

However, the list is not in a very useful order. You can add an ORDER BY clause to sort the column data in ascending order and make it easier to find that particular **po_num**, as Query Result 2-15 shows.

Query 2-15

```
SELECT po_num FROM orders
    ORDER BY po_num
```

Query Result 2-15

```
po_num

 278693
 278701
 2865
 429Q
 4745
 8006
 8052
 9270
 B77836
 B77890
 B77897
 B77930
 C3288
 DM354331
 KF2961
 LZ230
 MA003
 PC6782
 Q13557
 S22942
 W2286
 W9925
 Z55709
```

To select multiple columns from a table, list them in the select list in the SELECT clause. Query 2-16 shows that the order in which the columns are selected is the order in which they are produced, from left to right.

Query 2-16

```
SELECT paid_date, ship_date, order_date,
       customer_num, order_num, po_num
    FROM orders
    ORDER BY paid_date, order_date, customer_num
```

As shown in "Sorting on Multiple Columns" on page 2-15, you can use the ORDER BY clause to sort the data in ascending or descending order and perform nested sorts. Query Result 2-16 shows ascending order.

Query Result 2-16

```
paid_date  ship_date  order_date customer_num   order_num po_num

           05/30/1998 05/22/1998          106        1004 8006
                      05/30/1998          112        1006 Q13557
           06/05/1998 05/31/1998          117        1007 278693
           06/29/1998 06/18/1998          117        1012 278701
           07/12/1998 06/29/1998          119        1016 PC6782
           07/13/1998 07/09/1998          120        1017 DM354331
06/03/1998 05/26/1998 05/21/1998          101        1002 9270
06/14/1998 05/23/1998 05/22/1998          104        1003 B77890
06/21/1998 06/09/1998 05/24/1998          116        1005 2865
07/10/1998 07/03/1998 06/25/1998          106        1014 8052
07/21/1998 07/06/1998 06/07/1998          110        1008 LZ230
07/22/1998 06/01/1998 05/20/1998          104        1001 B77836
07/31/1998 07/10/1998 06/22/1998          104        1013 B77930
08/06/1998 07/13/1998 07/10/1998          121        1018 S22942
08/06/1998 07/16/1998 07/11/1998          122        1019 Z55709
08/21/1998 06/21/1998 06/14/1998          111        1009 4745
08/22/1998 06/29/1998 06/17/1998          115        1010 429Q
08/22/1998 07/25/1998 07/23/1998          124        1021 C3288
08/22/1998 07/30/1998 07/24/1998          127        1023 KF2961
08/29/1998 07/03/1998 06/18/1998          104        1011 B77897
08/31/1998 07/16/1998 06/27/1998          110        1015 MA003
09/02/1998 07/30/1998 07/24/1998          126        1022 W9925
09/20/1998 07/16/1998 07/11/1998          123        1020 W2286
```

When you use SELECT and ORDER BY on several columns in a table, you might find it helpful to use integers to refer to the position of the columns in the ORDER BY clause.The statements in Query 2-17 retrieve and display the same data, as Query Result 2-17 shows.

Query 2-17

```
SELECT customer_num, order_num, po_num, order_date
    FROM orders
    ORDER BY 4, 1

SELECT customer_num, order_num, po_num, order_date
    FROM orders
    ORDER BY order_date, customer_num
```

Query Result 2-17

```
customer_num   order_num po_num     order_date

         104        1001 B77836     05/20/1998
         101        1002 9270       05/21/1998
         104        1003 B77890     05/22/1998
         106        1004 8006       05/22/1998
         116        1005 2865       05/24/1998
         112        1006 Q13557     05/30/1998
         117        1007 278693     05/31/1998
         110        1008 LZ230      06/07/1998
         111        1009 4745       06/14/1998
         115        1010 429Q       06/17/1998
         104        1011 B77897     06/18/1998
         117        1012 278701     06/18/1998
         104        1013 B77930     06/22/1998
         106        1014 8052       06/25/1998
         110        1015 MA003      06/27/1998
         119        1016 PC6782     06/29/1998
         120        1017 DM354331   07/09/1998
         121        1018 S22942     07/10/1998
         122        1019 Z55709     07/11/1998
         123        1020 W2286      07/11/1998
         124        1021 C3288      07/23/1998
         126        1022 W9925      07/24/1998
         127        1023 KF2961     07/24/1998
```

You can include the DESC keyword in the ORDER BY clause when you assign integers to column names, as Query 2-18 shows.

Query 2-18

```
SELECT customer_num, order_num, po_num, order_date
    FROM orders
    ORDER BY 4 DESC, 1
```

In this case, data is first sorted in descending order by **order_date** and in ascending order by **customer_num**.

GLS

ORDER BY and Non-English Data

By default, Informix database servers use the U.S. English language environment, called a locale, for database data. The U.S. English locale specifies data sorted in code-set order. This default locale uses the ISO 8859-1 code set.

If your database contains non-English data, the ORDER BY clause should return data in the order appropriate to that language. Query 2-19 uses a SELECT statement with an ORDER BY clause to search the table, **abonnés**, and to order the selected information by the data in the **nom** column.

Query 2-19

```
SELECT numéro,nom,prénom
    FROM abonnés
    ORDER BY nom;
```

The collation order for the results of this query can vary, depending on the following system variations:

- Whether the **nom** column is CHAR or NCHAR data type. The database server sorts data in CHAR columns by the order the characters appear in the code set. The database server sorts data in NCHAR columns by the order the characters are listed in the collation portion of the locale. Store non-English data in NCHAR (or NVARCHAR) columns to obtain results sorted by the language.
- Whether the database server is using the correct non-English locale when it accesses the database. To use a non-English locale, you must set the CLIENT_LOCALE and DB_LOCALE environment variables to the appropriate locale name.

For Query 2-19 to return expected results, the **nom** column should be NCHAR data type in a database that uses a French locale. Other operations, such as less than, greater than, or equal to, are also affected by the user-specified locale. For more information on non-English data and locales, see the *Informix Guide to GLS Functionality*.

Query Result 2-19a and Query Result 2-19b show two sample sets of output.

Query Result 2-19a

```
numéro  nom         prénom

13612   Azevedo    Edouardo Freire
13606   Dupré      Michèle Françoise
13607   Hammer     Gerhard
13602   Hämmer     le Greta
13604   LaForêt    Jean-Noël
13610   LeMaître   Héloïse
13613   Llanero    Gloria Dolores
13603   Montaña    José Antonio
13611   Oatfield   Emily
13609   Tiramisù   Paolo Alfredo
13600   da Sousa   João Lourenço Antunes
13615   di GirolamoGiuseppe
13601   Ålesund    Sverre
13608   Étaix      Émile
13605   Ötker      Hans-Jürgen
13614   Øverst     Per-Anders
```

Query Result 2-19a follows the ISO 8859-1 code-set order, which ranks uppercase letters before lowercase letters and moves the names that start with an accented character (Ålesund, Étaix, Ötker, and Øverst) to the end of the list.

Query Result 2-19b

```
numéro    nom         prénom

13601     Ålesund     Sverre
13612     Azevedo     Edouardo Freire
13600     da Sousa    João Lourenço Antunes
13615     di Girolamo Giuseppe
13606     Dupré       Michèle Françoise
13608     Étaix       Émile
13607     Hammer      Gerhard
13602     Hämmer      le Greta
13604     LaForêt     Jean-Noël
13610     LeMaître    Héloïse
13613     Llanero     Gloria Dolores
13603     Montaña     José Antonio
13611     Oatfield    Emily
13605     Ötker       Hans-Jürgen
13614     Øverst      Per-Anders
13609     Tiramisù    Paolo Alfredo
```

Query Result 2-19b shows that when the appropriate locale file is referenced by the data server, names starting with non-English characters (Ålesund, Étaix, Ötker, and Øverst) are collated differently than they are in the ISO 8859-1 code set. They are sorted correctly for the locale. It does not distinguish between uppercase and lowercase letters.

Selecting Substrings

To select part of the value of a character column, include a *substring* in the select list. Suppose your marketing department is planning a mailing to your customers and wants a rough idea of their geographical distribution based on zip codes. You could write a query similar to the one Query 2-20 shows.

Query 2-20

```
SELECT zipcode[1,3], customer_num
    FROM customer
    ORDER BY zipcode
```

Query 2-20 uses a substring to select the first three characters of the **zipcode** column (which identify the state) and the full **customer_num**, and lists them in ascending order by zip code, as Query Result 2-20 shows.

Query Result 2-20

```
zipcode customer_num

021              125
080              119
085              122
198              121
322              123
604              127
740              124
802              126
850              128
850              120
940              105
940              112
940              113
940              115
940              104
940              116
940              110
940              114
940              106
940              108
940              117
940              111
940              101
940              109
941              102
943              103
943              107
946              118
```

Using the WHERE Clause

Add a WHERE clause to a SELECT statement if you want to see only those orders that a particular customer placed or the calls that a particular customer service representative entered.

You can use the WHERE clause to set up a *comparison condition* or a *join condition*. This section demonstrates only the first use. Join conditions are described in a later section and in the next chapter.

The set of rows returned by a SELECT statement is its *active set*. A *singleton* SELECT statement returns a single row. Use a *cursor* to retrieve multiple rows in an SQL API. See Chapter 5, "Programming with SQL," and Chapter 6, "Modifying Data Through SQL Programs."

Creating a Comparison Condition

The WHERE clause of a SELECT statement specifies the rows that you want to see. A comparison condition employs specific *keywords* and *operators* to define the search criteria.

For example, you might use one of the keywords BETWEEN, IN, LIKE, or MATCHES to test for equality, or the keywords IS NULL to test for null values. You can combine the keyword NOT with any of these keywords to specify the opposite condition.

The following table lists the *relational operators* that you can use in a WHERE clause in place of a keyword to test for equality.

Operator	Operation
=	equals
!= or <>	does not equal
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

For CHAR expressions, *greater than* means *after* in ASCII collating order, where lowercase letters are after uppercase letters, and both are after numerals. See the ASCII Character Set chart in the *Informix Guide to SQL: Syntax*. For DATE and DATETIME expressions, *greater than* means *later in time*, and for INTERVAL expressions, it means *of longer duration*.

Important: *You cannot use TEXT or BYTE columns in string expressions, except when you test for null values.*

You can use the preceding keywords and operators in a WHERE clause to create comparison-condition queries that perform the following actions:

- Include values
- Exclude values
- Find a range of values
- Find a subset of values
- Identify null values

To perform variable text searches using the criteria listed below, use the preceding keywords and operators in a WHERE clause to create comparison-condition queries:

- Exact-text comparison
- Single-character wildcards
- Restricted single-character wildcards
- Variable-length wildcards
- Subscripting

The following section contains examples that illustrate these types of queries.

Including Rows

Use the equal sign (=) relational operator to include rows in a WHERE clause, as Query 2-21 shows.

Query 2-21

```
SELECT customer_num, call_code, call_dtime, res_dtime
    FROM cust_calls
    WHERE user_id = 'maryj'
```

Query 2-21 returns the set of rows that Query Result 2-21 shows.

Query Result 2-21

```
customer_num call_code call_dtime       res_dtime

         106  D        1998-06-12 08:20 1998-06-12 08:25
         121  O        1998-07-10 14:05 1998-07-10 14:06
         127  I        1998-07-31 14:30
```

Excluding Rows

Use the relational operators != or <> to exclude rows in a WHERE clause.

Query 2-22 assumes that you are selecting from an ANSI-compliant database; the statements specify the *owner* or login name of the creator of the **customer** table. This qualifier is not required when the creator of the table is the current user, or when the database is not ANSI compliant. However, you can include the qualifier in either case. For a complete discussion of owner naming, see the *Informix Guide to SQL: Syntax*.

Query 2-22

```
SELECT customer_num, company, city, state
    FROM odin.customer
    WHERE state != 'CA'

SELECT customer_num, company, city, state
    FROM odin.customer
    WHERE state <> 'CA'
```

Both statements in Query 2-22 exclude values by specifying that, in the **customer** table that the user **odin** owns, the value in the **state** column should not be equal to CA, as Query Result 2-22 shows.

Query Result 2-22

```
customer_num  company              city           state

         119 The Triathletes Club Cherry Hill    NJ
         120 Century Pro Shop     Phoenix        AZ
         121 City Sports          Wilmington     DE
         122 The Sporting Life    Princeton      NJ
         123 Bay Sports           Jacksonville   FL
         124 Putnum's Putters     Bartlesville   OK
         125 Total Fitness Sports Brighton       MA
         126 Neelie's Discount Sp Denver         CO
         127 Big Blue Bike Shop   Blue Island    NY
         128 Phoenix College      Phoenix        AZ
```

Specifying A Range of Rows

Query 2-23 shows two ways to specify a range of rows in a WHERE clause.

Query 2-23

```
SELECT catalog_num, stock_num, manu_code, cat_advert
    FROM catalog
    WHERE catalog_num BETWEEN 10005 AND 10008

SELECT catalog_num, stock_num, manu_code, cat_advert
    FROM catalog
    WHERE catalog_num >= 10005 AND catalog_num <= 10008
```

Each statement in Query 2-23 specifies a range for **catalog_num** from 10005 through 10008, inclusive. The first statement uses keywords, and the second uses relational operators to retrieve the rows as Query Result 2-23 shows.

Query Result 2-23

```
catalog_num  10005
stock_num    3
manu_code    HSK
cat_advert   High-Technology Design Expands the Sweet Spot

catalog_num  10006
stock_num    3
manu_code    SHM
cat_advert   Durable Aluminum for High School and Collegiate              Athle
tes

catalog_num  10007
stock_num    4
manu_code    HSK
cat_advert   Quality Pigskin with Joe Namath Signature

catalog_num  10008
stock_num    4
manu_code    HRO
cat_advert   Highest Quality Football for High School
              and Collegiate Competitions
```

Although the **catalog** table includes a column with the BYTE data type, that column is not included in this SELECT statement because the output would show only the words <BYTE value> by the column name. You can write an SQL API application to display TEXT and BYTE values.

Excluding a Range of Rows

Query 2-24 uses the keywords NOT BETWEEN to exclude rows that have the character range 94000 through 94999 in the **zipcode** column, as Query Result 2-24 shows.

Query 2-24

```
SELECT fname, lname, company, city, state
    FROM customer
    WHERE zipcode NOT BETWEEN '94000' AND '94999'
    ORDER BY state
```

Query Result 2-24

```
fname           lname           company              city            state

Fred            Jewell          Century* Pro Shop    Phoenix         AZ
Frank           Lessor          Phoenix University   Phoenix         AZ
Eileen          Neelie          Neelie’s Discount Sp Denver          CO
Jason           Wallack         City Sports          Wilmington      DE
Marvin          Hanlon          Bay Sports           Jacksonville    FL
James           Henry           Total Fitness Sports Brighton        MA
Bob             Shorter         The Triathletes Club Cherry Hill     NJ
Cathy           O’Brian         The Sporting Life    Princeton       NJ
Kim             Satifer         Big Blue Bike Shop   Blue Island     NY
Chris           Putnum          Putnum’s Putters     Bartlesville    OK
```

Using a WHERE Clause to Find a Subset of Values

As shown in “Excluding Rows” on page 2-31, Query 2-25 also assumes the use of an ANSI-compliant database. The owner qualifier is in quotation marks to preserve the case sensitivity of the literal string.

Query 2-25

```
SELECT lname, city, state, phone
    FROM 'Aleta'.customer
    WHERE state = 'AZ' OR state = 'NJ'
    ORDER BY lname

SELECT lname, city, state, phone
    FROM 'Aleta'.customer
    WHERE state IN ('AZ', 'NJ')
    ORDER BY lname
```

Each statement in Query 2-25 retrieves rows that include the subset of AZ or NJ in the **state** column of the **Aleta.customer** table, as Query Result 2-25 shows.

Query Result 2-25

```
lname           city            state phone

Jewell          Phoenix         AZ    602-265-8754
Lessor          Phoenix         AZ    602-533-1817
O'Brian         Princeton       NJ    609-342-0054
Shorter         Cherry Hill     NJ    609-663-6079
```

Important: *You cannot test a TEXT or BYTE column with the IN keyword.*

In Query 2-26, an example of a query on an ANSI-compliant database, no quotation marks exist around the table owner name. Whereas the two statements in Query 2-25 searched the **Aleta.customer** table, Query 2-26 searches the table **ALETA.customer**, which is a different table, because of the way ANSI-compliant databases look at owner names.

Query 2-26

```
SELECT lname, city, state, phone
    FROM Aleta.customer
    WHERE state NOT IN ('AZ', 'NJ')
    ORDER BY state
```

Query 2-26 adds the keywords NOT IN, so the subset changes to exclude the subsets AZ and NJ in the **state** column. Query Result 2-26 shows the results in order of the **state** column.

Query Result 2-26

```
lname           city            state phone

Pauli           Sunnyvale       CA    408-789-8075
Sadler          San Francisco   CA    415-822-1289
Currie          Palo Alto       CA    415-328-4543
Higgins         Redwood City    CA    415-368-1100
Vector          Los Altos       CA    415-776-3249
Watson          Mountain View   CA    415-389-8789
Ream            Palo Alto       CA    415-356-9876
Quinn           Redwood City    CA    415-544-8729
Miller          Sunnyvale       CA    408-723-8789
Jaeger          Redwood City    CA    415-743-3611
Keyes           Sunnyvale       CA    408-277-7245
Lawson          Los Altos       CA    415-887-7235
Beatty          Menlo Park      CA    415-356-9982
Albertson       Redwood City    CA    415-886-6677
Grant           Menlo Park      CA    415-356-1123
Parmelee        Mountain View   CA    415-534-8822
Sipes           Redwood City    CA    415-245-4578
Baxter          Oakland         CA    415-655-0011
Neelie          Denver          CO    303-936-7731
Wallack         Wilmington      DE    302-366-7511
Hanlon          Jacksonville    FL    904-823-4239
Henry           Brighton        MA    617-232-4159
Satifer         Blue Island     NY    312-944-5691
Putnum          Bartlesville    OK    918-355-2074
```

Identifying Null Values

Use the IS NULL or IS NOT NULL option to check for null values. A null value represents either the absence of data or an unknown value. A null value is not the same as a zero or a blank.

Query 2-27 returns all rows that have a null **paid_date**, as Query Result 2-27 shows.

Query 2-27

```
SELECT order_num, customer_num, po_num, ship_date
    FROM orders
    WHERE paid_date IS NULL
    ORDER BY customer_num
```

Query Result 2-27

```
    order_num  customer_num  po_num   ship_date

         1004           106  8006     05/30/1998
         1006           112  Q13557
         1007           117  278693   06/05/1998
         1012           117  278701   06/29/1998
         1016           119  PC6782   07/12/1998
         1017           120  DM354331 07/13/1998
```

Forming Compound Conditions

To connect two or more comparison conditions, or *Boolean* expressions, use the *logical operators* AND, OR, and NOT. A Boolean expression evaluates as `true` or `false` or, if null values are involved, as `unknown`. You can use TEXT or BYTE objects in a Boolean expression only when you test for a null value.

In Query 2-28, the operator AND combines two comparison expressions in the WHERE clause.

Query 2-28

```
SELECT order_num, customer_num, po_num, ship_date
    FROM orders
    WHERE paid_date IS NULL
        AND ship_date IS NOT NULL
    ORDER BY customer_num
```

The query returns all rows that have a null **paid_date** *and* the ones that do not also have a null **ship_date**, as Query Result 2-28 shows.

Query Result 2-28

```
   order_num customer_num po_num     ship_date

        1004          106 8006       05/30/1998
        1007          117 278693     06/05/1998
        1012          117 278701     06/29/1998
        1016          119 PC6782     07/12/1998
        1017          120 DM354331   07/13/1998
```

Using Variable-Text Searches

You can use the keywords LIKE and MATCHES for *variable-text* queries that are based on substring searches of fields. Include the keyword NOT to indicate the opposite condition. The keyword LIKE is the ANSI standard, whereas MATCHES is an Informix extension.

Variable-text search strings can include the wildcards listed with LIKE or MATCHES in the following table.

Symbol	**Meaning**
LIKE	
%	Evaluates to zero or more characters
_	Evaluates to a single character
\	Escapes special significance of next character
MATCHES	
*	Evaluates to zero or more characters
?	Evaluates to a single character (except null)
[ ]	Evaluates to a single character or range of values
\	Escapes special significance of next character

You cannot test a TEXT or BYTE column with LIKE or MATCHES.

Using Exact-Text Comparisons

The following examples include a WHERE clause that searches for exact-text comparisons by using the keyword LIKE or MATCHES or the equal sign (=) relational operator. Unlike earlier examples, these examples illustrate how to query a table that is not in the current database. You can access a table that is not in the current database only if the table is part of an ANSI-compliant database.

Although the database used previously in this chapter is the demonstration database, the FROM clause in the following examples specifies the **manatee** table, created by the owner **bubba**, which resides in an ANSI-compliant database named **syzygy**. For more information on how to define tables that are not in the current database, see the *Informix Guide to SQL: Syntax*

Each statement in Query 2-29 retrieves all the rows that have the single word `helmet` in the **description** column as Query Result 2-29 shows.

Query 2-29

```
SELECT * FROM syzygy:bubba.manatee
    WHERE description = 'helmet'
    ORDER BY mfg_code

SELECT * FROM syzygy:bubba.manatee
    WHERE description LIKE 'helmet'
    ORDER BY mfg_code

SELECT * FROM syzygy:bubba.manatee
    WHERE description MATCHES 'helmet'
    ORDER BY mfg_code
```

Query Result 2-29

```
stock_no mfg_code  description     unit_price unit unit_type

     991 ANT       helmet             $222.00 case 4/case
     991 BKE       helmet             $269.00 case 4/case
     991 JSK       helmet             $311.00 each 4/case
     991 PRM       helmet             $234.00 case 4/case
     991 SHR       helmet             $245.00 case 4/case
```

Using a Single-Character Wildcard

The statements in Query 2-30 illustrate the use of a single-character wildcard in a WHERE clause. Further, they demonstrate a query on a table that is not in the current database. The **stock** table is in the database **sloth**. Beside being outside the current demonstration database, **sloth** is on a separate database server called **meerkat**.

For information about how to select tables from a database that is not the current database, see "Selecting Tables from a Database Other Than the Current Database" on page 2-101 in this manual, and the *Informix Guide to SQL: Syntax*.

Query 2-30

```
SELECT * FROM sloth@meerkat:stock
    WHERE manu_code LIKE '_R_'
        AND unit_price >= 100
    ORDER BY description, unit_price

SELECT * FROM sloth@meerkat:stock
    WHERE manu_code MATCHES '?R?'
        AND unit_price >= 100
    ORDER BY description, unit_price
```

Each statement in Query 2-30 retrieves only those rows for which the middle letter of the **manu_code** is R, as Query Result 2-30 shows.

Query Result 2-30

```
stock_num manu_code description     unit_price unit unit_descr

      205 HRO       3 golf balls       $312.00 case 24/case
        2 HRO       baseball           $126.00 case 24/case
        1 HRO       baseball gloves    $250.00 case 10 gloves/case
        7 HRO       basketball         $600.00 case 24/case
      102 PRC       bicycle brakes     $480.00 case 4 sets/case
      114 PRC       bicycle gloves     $120.00 case 10 pairs/case
        4 HRO       football           $480.00 case 24/case
      110 PRC       helmet             $236.00 case 4/case
      110 HRO       helmet             $260.00 case 4/case
      307 PRC       infant jogger      $250.00 each each
      306 PRC       tandem adapter     $160.00 each each
      308 PRC       twin jogger        $280.00 each each
      304 HRO       watch              $280.00 box  10/box
```

The comparison '_R_' (for LIKE) or '?R?' (for MATCHES) specifies, from left to right, the following items:

- Any single character
- The letter R
- Any single character

WHERE Clause with Restricted Single-Character Wildcard

Query 2-31 selects only those rows where the **manu_code** begins with A through H and returns the rows Query Result 2-31 shows. The class test '[A-H]' specifies any single letter from A through H, inclusive. No equivalent wildcard symbol exists for the LIKE keyword.

Query 2-31

```
SELECT * FROM stock
    WHERE manu_code MATCHES '[A-H]*'
    ORDER BY description, manu_code, unit_price
```

Query Result 2-31

```
stock_num manu_code description     unit_price unit unit_descr

      205 ANZ       3 golf balls       $312.00 case 24/case
      205 HRO       3 golf balls       $312.00 case 24/case
        2 HRO       baseball           $126.00 case 24/case
        3 HSK       baseball bat       $240.00 case 12/case
        1 HRO       baseball gloves    $250.00 case 10 gloves/case
        1 HSK       baseball gloves    $800.00 case 10 gloves/case
        7 HRO       basketball         $600.00 case 24/case
      .
      .
      .
      110 ANZ       helmet             $244.00 case 4/case
      110 HRO       helmet             $260.00 case 4/case
      110 HSK       helmet             $308.00 case 4/case
      .
      .
      .
      301 ANZ       running shoes       $95.00 each each
      301 HRO       running shoes       $42.50 each each
      313 ANZ       swim cap            $60.00 box  12/box
        6 ANZ       tennis ball         $48.00 case 24 cans/case
        5 ANZ       tennis racquet      $19.80 each each
        8 ANZ       volleyball         $840.00 case 24/case
        9 ANZ       volleyball net      $20.00 each each
      304 ANZ       watch              $170.00 box  10/box
      304 HRO       watch              $280.00 box  10/box
```

WHERE Clause with Variable-Length Wildcard

The statements in Query 2-32 use a wildcard at the end of a string to retrieve all the rows where the **description** begins with the characters `bicycle`.

Query 2-32

```
SELECT * FROM stock
    WHERE description LIKE 'bicycle%'
    ORDER BY description, manu_code

SELECT * FROM stock
    WHERE description MATCHES 'bicycle*'
    ORDER BY description, manu_code
```

Either statement returns the rows that Query Result 2-32 shows.

Query Result 2-32

```
stock_num manu_code description     unit_price unit unit_descr

      102 PRC       bicycle brakes     $480.00 case 4 sets/case
      102 SHM       bicycle brakes     $220.00 case 4 sets/case
      114 PRC       bicycle gloves     $120.00 case 10 pairs/case
      107 PRC       bicycle saddle      $70.00 pair pair
      106 PRC       bicycle stem        $23.00 each each
      101 PRC       bicycle tires       $88.00 box  4/box
      101 SHM       bicycle tires       $68.00 box  4/box
      105 PRC       bicycle wheels      $53.00 pair pair
      105 SHM       bicycle wheels      $80.00 pair pair
```

The comparison '`bicycle%`' or '`bicycle*`' specifies the characters `bicycle` followed by any sequence of zero or more characters. It matches `bicycle stem` with `stem` matched by the wildcard. It matches to the characters `bicycle` alone, if a row exists with that description.

Query 2-33 narrows the search by adding another comparison condition that excludes a **manu_code** of `PRC`.

Query 2-33

```
SELECT * FROM stock
    WHERE description LIKE 'bicycle%'
        AND manu_code NOT LIKE 'PRC'
    ORDER BY description, manu_code
```

The statement retrieves only the rows that Query Result 2-33 shows.

Query Result 2-33

```
stock_num manu_code description     unit_price unit unit_descr

      102 SHM       bicycle brakes     $220.00 case 4 sets/case
      101 SHM       bicycle tires       $68.00 box  4/box
      105 SHM       bicycle wheels      $80.00 pair pair
```

When you select from a large table and use an initial wildcard in the comparison string (such as '%cycle'), the query often takes longer to execute. Because indexes cannot be used, every row is searched.

GLS

MATCHES and Non-English Data

By default, Informix database servers use the U.S. English language environment, called a locale, for database data. This default locale uses the ISO 8859-1 code set. The U.S. English locale specifies that MATCHES will use code-set order.

If your database contains non-English data, the MATCHES clause should use the correct non-English code set for that language. Query 2-34 uses a SELECT statement with a MATCHES clause in the WHERE clause to search the table, **abonnés**, and to compare the selected information with the data in the **nom** column.

Query 2-34

```
SELECT numéro,nom,prénom
    FROM abonnés
    WHERE nom MATCHES '[E-P]*'
    ORDER BY nom;
```

The result of the comparison in this query is the same whether **nom** is A CHAR or NCHAR column. The database server uses the sort order that the locale specifies to determine what characters are in the range E through P. This behavior is an exception to the rule that the database server collates CHAR and VARCHAR columns in code-set order and NCHAR and NVARCHAR columns in the sort order that the locale specifies.

In Query Result 2-34a, the rows for Étaix, Ötker, and Øverst are not selected and listed because, with ISO 8859-1 code-set order, the accented first letter of each name is not in the `E` through `P` MATCHES range for the **nom** column.

Query Result 2-34a

```
numéro  nom        prénom

13607   Hammer     Gerhard
13602   Hämmer     Greta
13604   LaForêt    Jean-Noël
13610   LeMaître   Héloïse
13613   Llanero    Gloria Dolores
13603   Montaña    José Antonio
13611   Oatfield   Emily
```

The database server uses code-set order when the **nom** column is CHAR data type. It also uses localized ordering when the column is NCHAR data type, and you specify a nondefault locale.

In Query Result 2-34a, the rows for Étaix, Ötker, and Øverst *are* included in the list because the database server uses a locale-specific comparison.

Query Result 2-34b

```
numéro  nom       prénom

13608   Étaix     Émile
13607   Hammer    Gerhard
13602   Hämmer    Greta
13604   LaForêt   Jean-Noël
13610   LeMaître  Héloïse
13613   Llanero   Gloria Dolores
13603   Montaña   José Antonio
13611   Oatfield  Emily
13605   Ötker     Hans-Jürgen
13614   Øverst    Per-Anders
```

For more information on non-English data and locales, see the *Informix Guide to GLS Functionality*.

Comparing for Special Characters

Query 2-35 uses the keyword ESCAPE with LIKE or MATCHES so you can protect a special character from misinterpretation as a wildcard symbol.

Query 2-35

```
SELECT * FROM cust_calls
    WHERE res_descr LIKE '%!%%' ESCAPE '!'
```

The ESCAPE keyword designates an *escape character* (it is ! in this example) that protects the next character so that it is interpreted as data and not as a wildcard. In the example, the escape character causes the middle percent sign (%) to be treated as data. By using the ESCAPE keyword, you can search for occurrences of a percent sign (%) in the **res_descr** column by using the LIKE wildcard percent sign (%). The query retrieves the row that Query Result 2-35 shows.

Query Result 2-35

```
customer_num   116
call_dtime     1997-12-21 11:24
user_id        mannyn
call_code      I
call_descr     Second complaint from this customer! Received
               two cases right-handed outfielder gloves
               (1 HRO) instead of one case lefties.
res_dtime      1997-12-27 08:19
res_descr      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
```

Using Subscripting in a WHERE Clause

You can use *subscripting* in the WHERE clause of a SELECT statement to specify a range of characters or numbers in a column, as Query 2-36 shows.

Query 2-36

```
SELECT catalog_num, stock_num, manu_code, cat_advert,
       cat_descr
    FROM catalog
    WHERE cat_advert[1,4] = 'High'
```

The subscript `[1,4]` causes Query 2-36 to retrieve all rows in which the first four letters of the **cat_advert** column are `High`, as Query Result 2-36 shows.

Query Result 2-36

```
 catalog_num  10004
 stock_num    2
 manu_code    HRO
 cat_advert   Highest Quality Ball Available, from
              Hand-Stitching to the Robinson Signature
 cat_descr
Jackie Robinson signature ball. Highest professional quality, used by National
League.

 catalog_num  10005
 stock_num    3
 manu_code    HSK
 cat_advert   High-Technology Design Expands the Sweet Spot
 cat_descr
Pro-style wood. Available in sizes: 31, 32, 33, 34, 35.

 catalog_num  10008
 stock_num    4
 manu_code    HRO
 cat_advert   Highest Quality Football for High School and
              Collegiate Competitions
 cat_descr
NFL-style, pigskin.


 catalog_num  10012
 stock_num    6
 manu_code    SMT
 cat_advert   High-Visibility Tennis, Day or Night
 cat_descr
Soft yellow color for easy visibility in sunlight or
artificial light.

 catalog_num  10043
 stock_num    202
 manu_code    KAR
 cat_advert   High-Quality Woods Appropriate for High School
              Competitions or Serious Amateurs
 cat_descr
Full set of woods designed for precision control and
power performance.

 catalog_num  10045
 stock_num    204
 manu_code    KAR
 cat_advert   High-Quality Beginning Set of Irons
              Appropriate for High School Competitions
 cat_descr
Ideally balanced for optimum control. Nylon covered shaft.

 catalog_num  10068
 stock_num    310
 manu_code    ANZ
 cat_advert   High-Quality Kickboard
 cat_descr
White. Standard size.
```

Using a FIRST Clause to Select Specific Rows

You can include a FIRST clause in a SELECT statement to specify that the query returns only a specified number of the first rows that match the conditions of the SELECT statement. You include a number immediately following the FIRST keyword to specify the maximum number of rows that the query can return. The rows that the database server returns when you execute a SELECT statement with a FIRST clause might differ, depending on whether the statement also includes an ORDER BY clause.

You cannot use a FIRST clause when the SELECT statement is a subquery or part of a view definition.

For information about restrictions on use of the FIRST clause, see the description of the SELECT statement in the *Informix Guide to SQL: Syntax*.

FIRST Clause Without an ORDER BY Clause

If you do not include an ORDER BY clause in a SELECT statement with a FIRST clause, any rows that match the conditions of the SELECT statement might be returned. In other words, the database server determines which of the qualifying rows to return, and the query result can vary depending on the query plan that the optimizer chooses.

Query 2-37 uses the FIRST clause to return the first five rows from the **state** table.

Query 2-37

```
SELECT FIRST 5 *
    FROM state
```

Query Result 2-37

```
code sname

AK   Alaska
HI   Hawaii
CA   California
OR   Oregon
WA   Washington
```

You can use a FIRST clause when you simply want to know the names of all the columns, and the type of data that a table contains, or to test a query that otherwise would return many rows. Query 2-38 shows how to use the FIRST clause to return column values for the first row of a table.

Query 2-38

```
SELECT FIRST 1 *
    FROM orders
```

Query Result 2-38

```
order_num      1001
order_date     05/20/1998
customer_num   104
ship_instruct  express
backlog        n
po_num         B77836
ship_date      06/01/1998
ship_weight    20.40
ship_charge    $10.00
paid_date      07/22/1998
```

FIRST Clause with an ORDER BY Clause

You can include an ORDER BY clause in a SELECT statement with a FIRST clause to return rows that contain the highest or lowest values for a specified column. Query 2-38 shows a query that includes an ORDER BY clause to return (by alphabetical order) the first five states contained in the **state** table. Query 2-39, which is the same as Query 2-37 except for the ORDER BY clause, returns a different set of rows than Query 2-37.

Query 2-39

```
SELECT FIRST 5 *
    FROM state ORDER BY sname
```

Query Result 2-39

```
code sname

AL   Alabama
AK   Alaska
AZ   Arizona
AR   Arkansas
CA   California
```

Query 2-40 shows how to use a FIRST clause in a query with an ORDER BY clause to find the 10 most expensive items listed in the **stock** table.

Query 2-40

```
SELECT FIRST 10 description, unit_price
    FROM stock ORDER BY unit_price DESC
```

Query Result 2-40

```
description     unit_price

football           $960.00
volleyball         $840.00
baseball gloves    $800.00
18-spd, assmbld    $685.90
irons/wedge        $670.00
basketball         $600.00
12-spd, assmbld    $549.00
10-spd, assmbld    $499.99
football           $480.00
bicycle brakes     $480.00
```

AD/XP

FIRST Clause in a Union Query

With Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, you can also use the FIRST clause to select the first rows that result from a union query. Query 2-41 uses a FIRST clause to return the first five rows of a union between the **stock** and **items** tables.

Query 2-41

```
SELECT FIRST 5 DISTINCT stock_num, manu_code
    FROM stock
    WHERE unit_price < 55.00

UNION

SELECT stock_num, manu_code
    FROM items
    WHERE quantity > 3
```

Query Result 2-41

```
stock_num manu_code

311        SHM
9          ANZ
301        HRO
6          ANZ
204        KAR
```

Expressions and Derived Values

You are not limited to selecting columns by name. You can use the SELECT clause of a SELECT statement to perform computations on column data and to display information *derived* from the contents of one or more columns. To do this, list an *expression* in the select list.

An expression consists of a column name, a constant, a quoted string, a keyword, or any combination of these items connected by operators. It can also include host variables (program data) when the SELECT statement is embedded in a program.

Arithmetic Expressions

An arithmetic expression contains at least one of the *arithmetic operators* listed in the following table and produces a number.

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division

Important: *You cannot use TEXT or BYTE columns in arithmetic expressions.*

Arithmetic operations enable you to see the results of proposed computations without actually altering the data in the database. You can add an INTO TEMP clause to save the altered data in a temporary table for further reference, computations, or impromptu reports. Query 2-42 calculates a 7 percent sales tax on the **unit_price** column when the **unit_price** is $400 or more (but does not update it in the database).

Query 2-42

```
SELECT stock_num, description, unit, unit_descr,
        unit_price, unit_price * 1.07
    FROM stock
    WHERE unit_price >= 400
```

If you are using DB-Access or Relational Object Manager, the result is displayed in a column labeled *expression*, as Query Result 2-42 shows.

Query Result 2-42

```
stock_num description     unit unit_descr      unit_price   (expression)

        1 baseball gloves case 10 gloves/case     $800.00    $856.0000
        1 baseball gloves case 10 gloves/case     $450.00    $481.5000
        4 football        case 24/case            $960.00   $1027.2000
        4 football        case 24/case            $480.00    $513.6000
        7 basketball      case 24/case            $600.00    $642.0000
        8 volleyball      case 24/case            $840.00    $898.8000
      102 bicycle brakes  case 4 sets/case        $480.00    $513.6000
      111 10-spd, assmbld each each               $499.99    $534.9893
      112 12-spd, assmbld each each               $549.00    $587.4300
      113 18-spd, assmbld each each               $685.90    $733.9130
      203 irons/wedge     case 2 sets/case        $670.00    $716.9000
```

Query 2-43 calculates a surcharge of $6.50 on orders when the quantity ordered is less than 5.

Query 2-43

```
SELECT item_num, order_num, quantity,
       total_price, total_price + 6.50
    FROM items
    WHERE quantity < 5
```

If you are using DB-Access or Relational Object Manager, the result appears in a column labeled *expression*, as Query Result 2-43 shows.

Query Result 2-43

```
item_num   order_num quantity total_price  (expression)

       1        1001        1     $250.00      $256.50
       1        1002        1     $960.00      $966.50
       2        1002        1     $240.00      $246.50
       1        1003        1      $20.00       $26.50
       2        1003        1     $840.00      $846.50
       1        1004        1     $250.00      $256.50
       2        1004        1     $126.00      $132.50
       3        1004        1     $240.00      $246.50
       4        1004        1     $800.00      $806.50
      .
      .
      .
       1        1021        2      $75.00       $81.50
       2        1021        3     $225.00      $231.50
       3        1021        3     $690.00      $696.50
       4        1021        2     $624.00      $630.50
       1        1022        1      $40.00       $46.50
       2        1022        2      $96.00      $102.50
       3        1022        2      $96.00      $102.50
       1        1023        2      $40.00       $46.50
       2        1023        2     $116.00      $122.50
       3        1023        1      $80.00       $86.50
       4        1023        1     $228.00      $234.50
       5        1023        1     $170.00      $176.50
       6        1023        1     $190.00      $196.50
```

Query 2-44 calculates and displays in an *expression* column (if you are using DB-Access or Relational Object Manager) the interval between when the customer call was received (**call_dtime**) and when the call was resolved (**res_dtime**), in days, hours, and minutes.

Query 2-44

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime
    FROM cust_calls
    ORDER BY user_id
```

Query Result 2-44

```
customer_num user_id            call_code call_dtime         (expression)

         116 mannyn             I         1997-12-21 11:24        5 20:55
         116 mannyn             I         1997-11-28 13:34        0 03:13
         106 maryj              D         1998-06-12 08:20        0 00:05
         121 maryj              O         1998-07-10 14:05        0 00:01
         127 maryj              I         1998-07-31 14:30
         110 richc              L         1998-07-07 10:24        0 00:06
         119 richc              B         1998-07-01 15:00        0 17:21
```

Using Display Labels

You can assign a *display label* to a computed or derived data column to replace the default column header *expression*. In Query 2-42, Query 2-43, and Query 2-44, the derived data is shown in a column called `(expression)`. Query 2-45 also presents derived values, but the column that displays the derived values has the descriptive header `taxed`.

Query 2-45

```
SELECT stock_num, description, unit, unit_descr,
       unit_price, unit_price * 1.07 taxed
    FROM stock
    WHERE unit_price >= 400
```

Query Result 2-45 shows that the label `taxed` is assigned to the expression in the select list that displays the results of the operation `unit_price * 1.07`.

Query Result 2-45

```
stock_num description     unit unit_descr      unit_price        taxed

        1 baseball gloves case 10 gloves/case     $800.00    $856.0000
        1 baseball gloves case 10 gloves/case     $450.00    $481.5000
        4 football        case 24/case            $960.00   $1027.2000
        4 football        case 24/case            $480.00    $513.6000
        7 basketball      case 24/case            $600.00    $642.0000
        8 volleyball      case 24/case            $840.00    $898.8000
      102 bicycle brakes  case 4 sets/case        $480.00    $513.6000
      111 10-spd, assmbld each each               $499.99    $534.9893
      112 12-spd, assmbld each each               $549.00    $587.4300
      113 18-spd, assmbld each each               $685.90    $733.9130
      203 irons/wedge     case 2 sets/case        $670.00    $716.9000
```

In Query 2-46, the label **surcharge** is defined for the column that displays the results of the operation `total_price + 6.50`.

Query 2-46

```
SELECT item_num, order_num, quantity,
       total_price, total_price + 6.50 surcharge
    FROM items
    WHERE quantity < 5
```

The **surcharge** column is labeled in the output, as Query Result 2-46 shows.

Query Result 2-46

```
item_num   order_num quantity total_price    surcharge
      .
      .
      .
       2        1013        1      $36.00       $42.50
       3        1013        1      $48.00       $54.50
       4        1013        2      $40.00       $46.50
       1        1014        1     $960.00      $966.50
       2        1014        1     $480.00      $486.50
       1        1015        1     $450.00      $456.50
       1        1016        2     $136.00      $142.50
       2        1016        3      $90.00       $96.50
       3        1016        1     $308.00      $314.50
       4        1016        1     $120.00      $126.50
       1        1017        4     $150.00      $156.50
       2        1017        1     $230.00      $236.50
       .
       .
       .
```

Query 2-47 assigns the label **span** to the column that displays the results of subtracting the DATETIME column **call_dtime** from the DATETIME column **res_dtime**.

Query 2-47

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime span
    FROM cust_calls
    ORDER BY user_id
```

The **span** column is labeled in the output, as Query Result 2-47 shows.

Query Result 2-47

```
customer_num user_id          call_code call_dtime            span

         116 mannyn           I         1997-12-21 11:24      5 20:55
         116 mannyn           I         1997-11-28 13:34      0 03:13
         106 maryj            D         1998-06-12 08:20      0 00:05
         121 maryj            O         1998-07-10 14:05      0 00:01
         127 maryj            I         1998-07-31 14:30
         110 richc            L         1998-07-07 10:24      0 00:06
         119 richc            B         1998-07-01 15:00      0 17:21
```

CASE Expressions

A CASE expression is a conditional expression, which is similar to the concept of the CASE statement in programming languages. You can use a CASE expression when you wish to change the way data is represented. The CASE expression allows a statement to return one of several possible results, depending on which of several condition tests evaluates to TRUE.

Consider a column that represents martial status numerically as 1, 2, 3, 4 with the corresponding meaning single, married, divorced, widowed. In some cases, you might prefer to store the short values (1,2,3,4) for database efficiency, but employees in human resources might prefer the more descriptive values (single, married, divorced, widowed). The CASE expression makes such conversions between different sets of values easy.

The following example shows a CASE statement with multiple WHEN clauses that returns more descriptive values for the **manu_code** column of the **stock** table. If none of the WHEN conditions is true, NULL is the default result. (You can omit the ELSE NULL clause.)

```
SELECT
    CASE
        WHEN manu_code = "HRO" THEN "Hero"
        WHEN manu_code = "SHM" THEN "Shimara"
        WHEN manu_code = "PRC" THEN "ProCycle"
        WHEN manu_code = "ANZ" THEN "Anza"
        ELSE NULL
    END
FROM stock;
```

You must include at least one WHEN clause within the CASE expression; subsequent WHEN clauses and the ELSE clause are optional. If no WHEN condition evaluates to true, the resulting value is null. You can use the IS NULL expression to handle null results. For information on handling null values, see the *Informix Guide to SQL: Syntax*.

The following example shows a simple CASE expression that returns a character string value to flag any orders from the **orders** table that have not been shipped to the customer:

Query 2-48

```
        CASE
            WHEN ship_date IS NULL
            THEN "order not shipped"
        END
FROM orders;
```

Query Result 2-48

```
order_num order_date (expression)

1001      05/20/1998
1002      05/21/1998
1003      05/22/1998
1004      05/22/1998
1005      05/24/1998
1006      05/30/1998 order not shipped
1007      05/31/1998
1008      06/07/1998
1009      06/14/1998
1010      06/17/1998
1011      06/18/1998
1012      06/18/1998
1013      06/22/1998
1014      06/25/1998
1015      06/27/1998
1016      06/29/1998
1017      07/09/1998
1018      07/10/1998
1019      07/11/1998
1020      07/11/1998
1021      07/23/1998
1022      07/24/1998
1023      07/24/1998
```

For information about how to use the CASE expression to update a column, see "Using a CASE Expression to Update a Column" on page 4-16.

Sorting on Derived Columns

When you want to use ORDER BY as an expression, you can use either the display label assigned to the expression, or an integer, as Query 2-49 shows.

Query 2-49

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime span
    FROM cust_calls
    ORDER BY span
```

Query 2-49 retrieves the same data from the **cust_calls** table as Query 2-47. In Query 2-49, the ORDER BY clause causes the data to be displayed in ascending order of the derived values in the **span** column, as Query Result 2-49 shows.

Query Result 2-49

```
customer_num user_id          call_code call_dtime              span
         127 maryj            I         1998-07-31 14:30
         121 maryj            O         1998-07-10 14:05        0 00:01
         106 maryj            D         1998-06-12 08:20        0 00:05
         110 richc            L         1998-07-07 10:24        0 00:06
         116 mannyn           I         1997-11-28 13:34        0 03:13
         119 richc            B         1998-07-01 15:00        0 17:21
         116 mannyn           I         1997-12-21 11:24        5 20:55
```

Query 2-50 uses an integer to represent the result of the operation `res_dtime - call_dtime` and retrieves the same rows that appear in Query Result 2-49.

Query 2-50

```
SELECT customer_num, user_id, call_code,
       call_dtime, res_dtime - call_dtime span
    FROM cust_calls
    ORDER BY 5
```

Using Functions in SELECT Statements

In addition to column names and operators, an expression can also include one or more functions. This section describes how to use aggregate functions, time functions, conversion functions, string manipulation functions, and other functions.

For information about the syntax of the following SQL functions and other SQL functions, see the Expressions segment in the *Informix Guide to SQL: Syntax*.

Aggregate Functions

All Informix database servers support the following *aggregate* functions:

- AVG
- COUNT
- MAX
- MIN
- RANGE
- STDEV

- SUM
- VARIANCE

Aggregate functions take on values that depend on all the rows selected and return information about rows, not the rows themselves.

Important: *You cannot use aggregate functions with TEXT or BYTE columns.*

Aggregates are often used to summarize information about groups of rows in a table. This use is discussed in Chapter 3, "Composing Advanced SELECT Statements." When you apply an aggregate function to an entire table, the result contains a single row that summarizes all of the selected rows.

Using the COUNT Function

Query 2-51 counts and displays the total number of rows in the **stock** table.

Query 2-51

```
SELECT COUNT(*)
    FROM stock
```

Query Result 2-51

```
(count(*))

        73
```

Query 2-52 includes a WHERE clause to count specific rows in the **stock** table; in this case, only those rows that have a **manu_code** of `SHM`.

Query 2-52

```
SELECT COUNT (*)
    FROM stock
    WHERE manu_code = 'SHM'
```

Query Result 2-52

```
(count(*))

        17
```

By including the keyword DISTINCT (or its synonym UNIQUE) and a column name in Query 2-53, you can tally the number of different manufacturer codes in the **stock** table.

Query 2-53

```
SELECT COUNT (DISTINCT manu_code)
    FROM stock
```

Query Result 2-53

```
      (count)

        9
```

Using the AVG Function

Query 2-54 computes the average **unit_price** of all rows in the **stock** table.

Query 2-54

```
SELECT AVG (unit_price)
    FROM stock
```

Query Result 2-54

```
       (avg)

     $197.14
```

Query 2-55 computes the average **unit_price** of just those rows in the **stock** table that have a **manu_code** of SHM.

Query 2-55

```
SELECT AVG (unit_price)
    FROM stock
    WHERE manu_code = 'SHM'
```

Query Result 2-55

```
        (avg)

      $204.93
```

Using the MAX and MIN Functions

You can combine aggregate functions in the same SELECT statement. For example, you can include both the MAX and the MIN functions in the select list, as Query 2-56 shows.

Query 2-56

```
SELECT MAX (ship_charge), MIN (ship_charge)
    FROM orders
```

Query 2-56 finds and displays both the highest and lowest **ship_charge** in the **orders** table, as Query Result 2-56 shows.

Query Result 2-56

```
 (max)     (min)

$25.20     $5.00
```

Using the SUM Function

Query 2-57 calculates the total **ship_weight** of orders that were shipped on July 13, 1998.

Query 2-57

```
SELECT SUM (ship_weight)
    FROM orders
    WHERE ship_date = '07/13/1998'
```

Query Result 2-57

```
(sum)

130.5
```

Using the RANGE Function

The RANGE function computes the range for a sample of a population. It computes the difference between the maximum and the minimum values.

You can apply the RANGE function only to numeric columns. Query 2-58 finds the range of prices for items in the **stock** table:

Query 2-58

```
SELECT RANGE(unit_price) FROM stock
```

Query Result 2-58

```
(range)

955.50
```

As with other aggregates, the RANGE function applies to the rows of a group when the query includes a GROUP BY clause, which Query 2-59 shows.

Query 2-59

```
SELECT RANGE(unit_price) FROM stock
GROUP BY manu_code
```

Query Result 2-59

```
(range)

820.20
595.50
720.00
225.00
632.50
0.00
460.00
645.90
425.00
```

Using the STDEV Function

The STDEV function computes the standard deviation for a sample of a population. It is the square root of the VARIANCE function.

You can apply the STDEV function only to numeric columns. The following query finds the standard deviation on a population:

```
SELECT STDEV(age) FROM u_pop WHERE u_pop.age > 0
```

As with the other aggregates, the STDEV function applies to the rows of a group when the query includes a GROUP BY clause, as shown in the following example:

```
SELECT STDEV(age) FROM u_pop
    GROUP BY birth
    WHERE STDEV(age) > 0
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the STDEV function returns a null for that column.

For more information about the STDEV function, see the Expression segment of the *Informix Guide to SQL: Syntax*.

Using the VARIANCE Function

The VARIANCE function returns the variance for a sample of values as an unbiased estimate of the variance of the population. It computes the following value:

```
(SUM(Xi**2) - (SUM(Xi)**2)/N)/(N-1)
```

In this example, *Xi* is each value in the column and *N* is the total number of values in the column. You can apply the VARIANCE function only to numeric columns. The following query finds the variance on a population:

```
SELECT VARIANCE(age) FROM u_pop WHERE u_pop.age > 0
```

As with the other aggregates, the VARIANCE function applies to the rows of a group when the query includes a GROUP BY clause, which the following example shows:

```
SELECT VARIANCE(age) FROM u_pop
    GROUP BY birth
    WHERE VARIANCE(age) > 0
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the VARIANCE function returns a null for that column.

For more information about the VARIANCE function, see the Expression segment of the *Informix Guide to SQL: Syntax*.

Applying Functions to Expressions

Query 2-60 shows how you can apply functions to expressions, and you can supply display labels for their results.

Query 2-60

```
SELECT MAX (res_dtime - call_dtime) maximum,
    MIN (res_dtime - call_dtime) minimum,
    AVG (res_dtime - call_dtime) average
    FROM cust_calls
```

Query 2-60 finds and displays the maximum, minimum, and average amount of time (in days, hours, and minutes) between the reception and resolution of a customer call and labels the derived values appropriately. Query Result 2-60 shows these amounts of time.

Query Result 2-60

```
maximum         minimum         average

5 20:55         0 00:01         1 02:56
```

Time Functions

You can use the *time* functions DAY, MDY, MONTH, WEEKDAY, and YEAR in either the SELECT clause or the WHERE clause of a query. These functions return a value that corresponds to the expressions or arguments that you use to call the function. You can also use the CURRENT function to return a value with the current date and time, or use the EXTEND function to adjust the precision of a DATE or DATETIME value.

Using DAY and CURRENT Functions

Query 2-61 returns the day of the month for the **call_dtime** and **res_dtime** columns in two *expression* columns.

Query 2-61

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
    FROM cust_calls
```

Query Result 2-61

```
customer_num  (expression) (expression)

         106            12           12
         110             7            7
         119             1            2
         121            10           10
         127            31
         116            28           28
         116            21           27
```

Query 2-62 uses the DAY and CURRENT functions to compare column values to the current day of the month. It selects only those rows where the value is earlier than the current day.

Query 2-62

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
    FROM cust_calls
    WHERE DAY (call_dtime) < DAY (CURRENT)
```

Query Result 2-62

```
customer_num  (expression) (expression)

         106            12           12
         110             7            7
         119             1            2
         121            10           10
```

Query 2-63 shows another use of the CURRENT function, selecting rows where the day is earlier than the current one.

Query 2-63

```
SELECT customer_num, call_code, call_descr
    FROM cust_calls
    WHERE call_dtime < CURRENT YEAR TO DAY
```

Query Result 2-63

```
customer_num  106
call_code     D
call_descr    Order was received, but two of the cans of ANZ tennis balls
              within the case were empty

customer_num  116
call_code     I
call_descr    Received plain white swim caps (313 ANZ) instead of navy with
              team logo (313 SHM)

customer_num  116
call_code     I
call_descr    Second complaint from this customer! Received two cases
              right-handed outfielder gloves (1 HRO) instead of one case
              lefties.
```

Using the MONTH Function

Query 2-64 uses the MONTH function to extract and show what month the customer call was received and resolved, and it uses display labels for the resulting columns. However, it does not make a distinction between years.

Query 2-64

```
SELECT customer_num,
    MONTH (call_dtime) call_month,
    MONTH (res_dtime) res_month
    FROM cust_calls
```

Query Result 2-64

```
customer_num  call_month   res_month

         106           6           6
         110           7           7
         119           7           7
         121           7           7
         127           7
         116          11          11
         116          12          12
```

Query 2-65 uses the MONTH function plus DAY and CURRENT to show what month the customer-call was received and resolved if DAY is earlier than the current day.

Query 2-65

```
SELECT customer_num,
    MONTH (call_dtime) called,
    MONTH (res_dtime) resolved
    FROM cust_calls
    WHERE DAY (res_dtime) < DAY (CURRENT)
```

Query Result 2-65

```
customer_num    called   resolved

         106         6          6
         119         7          7
         121         7          7
```

Using the WEEKDAY Function

Query 2-66 uses the WEEKDAY function to indicate which day of the week calls are received and resolved (0 represents Sunday, 1 is Monday, and so on), and the expression columns are labeled.

Query 2-66

```
SELECT customer_num,
    WEEKDAY (call_dtime) called,
    WEEKDAY (res_dtime) resolved
    FROM cust_calls
    ORDER BY resolved
```

Query Result 2-66

```
customer_num    called   resolved

         127         3
         110         0          0
         119         1          2
         121         3          3
         116         3          3
         106         3          3
         116         5          4
```

Query 2-67 uses the COUNT and WEEKDAY functions to count how many calls were received on a weekend. This kind of statement can give you an idea of customer-call patterns or indicate whether overtime pay might be required.

Query 2-67

```
SELECT COUNT(*)
    FROM cust_calls
    WHERE WEEKDAY (call_dtime) IN (0,6)
```

Query Result 2-67

```
(count(*))

         4
```

Using the YEAR Function

Query 2-68 retrieves rows where the **call_dtime** is earlier than the beginning of the current year.

Query 2-68

```
SELECT customer_num, call_code,
    YEAR (call_dtime) call_year,
    YEAR (res_dtime) res_year
    FROM cust_calls
    WHERE YEAR (call_dtime) < YEAR (TODAY)
```

Query Result 2-68

```
customer_num call_code call_year res_year

         116 I              1997     1997
         116 I              1997     1997
```

Formatting DATETIME Values

In Query 2-69, the EXTEND function displays only the specified subfields to restrict the two DATETIME values.

Query 2-69

```
SELECT customer_num,
    EXTEND (call_dtime, month to minute) call_time,
    EXTEND (res_dtime, month to minute) res_time
    FROM cust_calls
    ORDER BY res_time
```

Query Result 2-69 returns the month-to-minute range for the columns labeled **call_time** and **res_time** and gives an indication of the work load.

Query Result 2-69

```
customer_num call_time   res_time

         127 07-31 14:30
         106 06-12 08:20 06-12 08:25
         119 07-01 15:00 07-02 08:21
         110 07-07 10:24 07-07 10:30
         121 07-10 14:05 07-10 14:06
         116 11-28 13:34 11-28 16:47
         116 12-21 11:24 12-27 08:19
```

IDS

Date-Conversion Functions

The following conversion functions convert between date and character values:

- DATE
- TO_CHAR
- TO_DATE

You can use a date-conversion function anywhere you use an expression.

Using the DATE Function

The DATE function converts a character string to a DATE value. In Query 2-70, the DATE function converts a character string to a DATE value to allow for comparisons with DATETIME values. The query retrieves DATETIME values only when **call_dtime** is later than the specified DATE.

Query 2-70

```
SELECT customer_num, call_dtime, res_dtime
    FROM cust_calls
    WHERE call_dtime > DATE ('12/31/97')
```

Query Result 2-70

```
customer_num call_dtime       res_dtime

         106 1998-06-12 08:20 1998-06-12 08:25
         110 1998-07-07 10:24 1998-07-07 10:30
         119 1998-07-01 15:00 1998-07-02 08:21
         121 1998-07-10 14:05 1998-07-10 14:06
         127 1998-07-31 14:30
```

Query 2-71 converts DATETIME values to DATE format and displays the values, with labels, only when **call_dtime** is greater than or equal to the specified date.

Query 2-71

```
SELECT customer_num,
    DATE (call_dtime) called,
    DATE (res_dtime) resolved
    FROM cust_calls
    WHERE call_dtime >= DATE ('1/1/98')
```

Query Result 2-71

```
customer_num called       resolved

         106 06/12/1998   06/12/1998
         110 07/07/1998   07/07/1998
         119 07/01/1998   07/02/1998
         121 07/10/1998   07/10/1998
         127 07/31/1998
```

Using the TO_CHAR Function

The TO_CHAR function converts DATETIME (OR DATE) values to character string values. The TO_CHAR function evaluates a DATETIME value according to the date-formatting directive that you specify and returns a NVARCHAR value. For a complete list of the supported date-formatting directives, see the description of the **GL_DATETIME** environment variable in the *Informix Guide to GLS Functionality*.

Query 2-73 uses the TO_CHAR function to convert a DATETIME value to a more readable character string.

Query 2-72

```
SELECT customer_num,
     TO_CHAR(call_dtime, "%A %B %d %Y") call_date
    FROM cust_call
    WHERE call_code = "B"
```

Query Result 2-72

```
customer_num  119
call_date     Friday July 01 1998
```

Query 2-73 uses the TO_CHAR function to convert DATE values to more readable character strings.

Query 2-73

```
SELECT order_num,
     TO_CHAR(ship_date,"%A %B %d %Y") date_shipped
    FROM orders
    WHERE paid_date IS NULL
```

Query Result 2-73

```
order_num     1004
date_shipped  Monday May 30 1998

order_num     1006
date_shipped

order_num     1007
date_shipped  Sunday June 05 1998

order_num     1012
date_shipped  Wednesday June 29 1998

order_num     1016
date_shipped  Tuesday July 12 1998

order_num     1017
date_shipped  Wednesday July 13 1998
```

Using the TO_DATE Function

The TO_DATE function accepts an argument of a character data type and converts this value to a DATETIME value. The TO_DATE function evaluates a character string according to the date-formatting directive that you specify and returns a DATETIME value. For a complete list of the supported date-formatting directives, see the description of the **GL_DATETIME** environment variable in the *Informix Guide to GLS Functionality*.

Query 2-74 uses the TO_DATE function to convert character string values to DATETIME values whose format you specify.

Query 2-74

```
SELECT customer_num, call_descr
    FROM cust_calls
    WHERE call_dtime = TO_DATE("1998-07-07 10:24",
    "%Y-%m-%d %H:%M").
```

Query Result 2-74

```
customer_num   110

call_descr     Order placed one month ago (6/7) not received.
```

You can use the DATE or TO_DATE function to convert a character string to a DATE value. One advantage of the TO_DATE function is that it allows you to specify a format for the value returned. (You can use the TO_DATE function, which always returns a DATETIME value, to convert a character string to a DATE value because the database server implicitly handles conversions between DATE and DATETIME values.)

Query 2-75 uses the TO_DATE function to convert character string values to DATE values whose format you specify.

Query 2-75

```
SELECT order_num, paid_date
    FROM orders
    WHERE order_date = TO_DATE( "6/7/98", "%m/%d/%iY")
```

Query Result 2-75

```
order_num  paid_date

1008       07/21/1998
```

IDS

String Manipulation Functions

String manipulation functions accept arguments of type CHAR, NCHAR, VARCHAR, NVARCHAR, or LVARCHAR. You can use a string manipulation function anywhere you use an expression.

The following functions convert between upper and lower case letters in a character string:

- LOWER
- UPPER
- INITCAP

The following functions manipulate character strings in various ways:

- REPLACE
- SUBSTR
- SUBSTRING
- LPAD
- RPAD

Using the LOWER Function

You can use the LOWER function to replace every upper case letter in a character string with a lower case letter. The LOWER function accepts an argument of a character data type and returns a value of the same data type as the argument you specify.

Query 2-76 uses the LOWER function to convert any upper case letters in a character string to lower case letters.

Query 2-76

```
SELECT manu_code, LOWER(manu_code)
    FROM items
    WHERE order_num = 1018.
```

Query Result 2-76

```
manu_code  (expression)

PRC         prc
KAR         kar
PRC         prc
SMT         smt
HRO         hro
```

Using the UPPER Function

You can use the UPPER function to replace every lower case letter in a character string with an upper case letter. The UPPER function accepts an argument of a character data type and returns a value of the same data type as the argument you specify.

Query 2-77 uses the UPPER function to convert any upper case letters in a character string to lower case letters.

Query 2-77

```
SELECT call_code, UPPER(code_descr)
    FROM call_type
```

Query Result 2-77

```
call_code  (expression)

B           BILLING ERROR
D           DAMAGED GOODS
I           INCORRECT MERCHANDISE SENT
L           LATE SHIPMENT
O           OTHER
```

Using the INITCAP Function

You can use the INITCAP function to replace the first letter of every word in a character string with an upper case letter. The INITCAP function assumes a new word whenever the function encounters a letter that is preceded by any character other than a letter. The INITCAP function accepts an argument of a character data type and returns a value of the same data type as the argument you specify.

Query 2-78 uses the INITCAP function to convert the first letter of every word in a character string to an upper case letter.

Query 2-78

```
SELECT INITCAP(description)
    FROM stock
    WHERE manu_code = "ANZ"
```

Query Result 2-78

```
(expression)

3 Golf Balls
Golf Shoes
Helmet
Kick Board
Running Shoes
Swim Cap
Tennis Ball
Tennis Racquet
Volleyball
Volleyball Net
Watch
```

Using the REPLACE Function

You can use the REPLACE function to replace a certain set of characters in a character string with other characters.

In Query 2-79 the REPLACE function replaces the **unit** column value `each` with `item` for every row that the query returns. The first argument of the REPLACE function is the expression to be evaluated. The second argument specifies the characters that you wish to replace. The third argument specifies a new character string to replace the characters removed.

Query 2-79

```
SELECT stock_num, REPLACE(unit,"each", "item") cost_per,
unit_price
    FROM stock
    WHERE manu_code = "HRO"
```

Query Result 2-79

```
stock_num      cost_per    unit_price

1              case$250.00
2              case$126.00
4              case$480.00
7              case$600.00
110            case$260.00
205            case$312.00
301            item$ 42.50
302            item$  4.50
304            box $280.00
305            case$ 48.00
309            case$ 40.00
312            box $ 72.00
```

Using the SUBSTRING and SUBSTR Functions

You can use the SUBSTRING and SUBSTR functions to return a portion of a character string. You specify the *start position* and *length* (optional) to determine which portion of the character string the function returns.

Using the SUBSTRING Function

You can use the SUBSTRING function to return some portion of a character string. You specify the *start position* and *length* (optional) to determine which portion of the character string the function returns. You can specify a positive or negative number for the start position. A start position of 1 specifies that the SUBSTRING function begins from the first position in the string. When the start position is zero (0) or a negative number, the SUBSTRING function counts backward from the beginning of the string.

Query 2-80 shows an example of the SUBSTRING function, which returns the first four characters for any **sname** column values that the query returns. In this example, the SUBSTRING function starts at the beginning of the string and returns four characters counting forward from the start position.

Query 2-80

```
SELECT sname, SUBSTRING(sname FROM 1 FOR 4)
    FROM state
    WHERE code = "AZ"
```

Query Result 2-80

```
sname          (expression)

Arizona        Ariz
```

In Query 2-81 the SUBSTRING function specifies a start position of 6 but does not specify the length. The function returns a character string that extends from the sixth position to the end of the string.

Query 2-81

```
SELECT sname, SUBSTRING(sname FROM 6)
    FROM state
    WHERE code = "WV"
```

Query Result 2-81

```
sname          (expression)

West Virginia  Virginia
```

In Query 2-82, the SUBSTRING function returns only the first character for any **sname** column value that the query returns. For the SUBSTRING function, a start position of -2 counts backwards three positions (0, -1, -2) from the start position of the string (for a start position of 0 the function counts backward one position from the beginning of the string).

Query 2-82

```
SELECT sname, SUBSTRING(sname FROM -2 FOR 4)
    FROM state
    WHERE code = "AZ"
```

Query Result 2-82

```
sname          (expression)

Arizona        A
```

Using the SUBSTR Function

The SUBSTR function serves the same purpose as the SUBSTRING function, but the syntax of the two functions differs.

To return a portion of a character string, you specify the *start position* and *length* (optional) to determine which portion of the character string the SUBSTR function returns. The start position that you specify for the SUBSTR function can be a positive or a negative number. However, the SUBSTR function treats a negative number in the start position differently than does the SUBSTRING function. When the start position is a negative number, the SUBSTR function counts backward from the end of the character string, which depends of the length of the string, not the character length of a word or visible characters that the string contains. The SUBSTR function recognizes zero (0) or 1 in the start position as the first position in the string.

Query 2-83 shows an example of the SUBSTR function that includes a negative number for the start position. Given a start position of -15, the SUBSTR function counts backwards 15 positions from the end of the string to find the start position and then returns the next five characters.

Query 2-83

```
SELECT sname, SUBSTR(sname, -15, 5)
    FROM state
    WHERE code = "CA"
```

Query Result 2-83

```
sname          (expression)

California     Calif
```

To use a negative number for the start position, you need to know the length of the value that is evaluated. The **sname** column is defined as CHAR(15), so a SUBSTR function that accepts an argument of type **sname** can use a start position of 0, 1, or -15 for the function to return a character string beginning from the first position in the string.

Query 2-84 returns the same result as Query 2-83

Query 2-84

```
SELECT sname, SUBSTR(sname, 1, 5)
    FROM state
    WHERE code = "CA
```

Using the LPAD Function

You use the LPAD function to return a copy of a string that has been left padded with a sequence of characters that are repeated as many times as necessary or truncated, depending on the specified length of the padded portion of the string. You specify the source string, the length of the string to be returned, and the character string to serve as padding.

The data type of the source string and the character string that serves as padding can be any data type that converts to VARCHAR or NVARCHAR.

Query 2-83 shows an example of the LPAD function with a specified a length of 21 bytes. Because the source string has a length of 15 bytes (**sname** is defined as CHAR(15)) LPAD pads the first 6 positions to the left of the source string.

Query 2-85

```
SELECT sname, LPAD(sname, 21, "-")
    FROM state
    WHERE code = "CA"
```

Query Result 2-85

```
sname          (expression)

California     ------California
```

Using the RPAD Function

You use the RPAD function to return a copy of a string that has been left padded with a sequence of characters that are repeated as many times as necessary or truncated, depending on the specified length of the padded portion of the string. You specify the source string, the length of the string to be returned, and the character string to serve as padding.

The data type of the source string and the character string that serves as padding can be any data type that converts to VARCHAR or NVARCHAR.

Query 2-86 shows an example of the RPAD function with a specified a length of 21 bytes. Because the source string has a length of 15 bytes (**sname** is defined as CHAR(15)) the RPAD function pads the first 6 positions to the right of the source string.

Query 2-86

```
SELECT sname, RPAD(sname, 21, "-")
    FROM state
    WHERE code = "WV"
```

Query Result 2-86

```
sname          (expression)

West Virginia  West Virginia  ------
```

Other Functions

You can also use the LENGTH, USER, CURRENT, and TODAY functions anywhere in an SQL expression that you would use a constant. In addition, you can include the DBSERVERNAME function in a SELECT statement to display the name of the database server where the current database resides.

You can use these functions to select an expression that consists entirely of constant values or an expression that includes column data. In the first instance, the result is the same for all rows of output.

In addition, you can use the HEX function to return the hexadecimal encoding of an expression, the ROUND function to return the rounded value of an expression, and the TRUNC function to return the truncated value of an expression.

For more information on the preceding functions, see the *Informix Guide to SQL: Syntax*.

Using the LENGTH Function

In Query 2-87, the LENGTH function calculates the number of bytes in the combined **fname** and **lname** columns for each row where the length of **company** is greater than 15.

Query 2-87

```
SELECT customer_num,
    LENGTH (fname) + LENGTH (lname) namelength
    FROM customer
    WHERE LENGTH (company) > 15
```

Query Result 2-87

```
customer_num   namelength

         101           11
         105           13
         107           11
         112           14
         115           11
         118           10
         119           10
         120           10
         122           12
         124           11
         125           10
         126           12
         127           10
         128           11
```

Although the LENGTH function might not be useful when you work with DB-Access or Relational Object Manager, it can be important to determine the string length for programs and reports. LENGTH returns the clipped length of a CHARACTER or VARCHAR string and the full number of bytes in a TEXT or BYTE string.

Using the USER Function

You can use the USER function when you want to define a restricted view of a table that contains only your rows. For information about how to create views, see the *Informix Guide to Database Design and Implementation* and the GRANT and CREATE VIEW statements in the *Informix Guide to SQL: Syntax*.

Query 2-88a specifies the USER function and the **cust_calls** table.

Query 2-88a

```
SELECT USER FROM cust_calls
```

Query 2-88b returns the user name (login account name) of the user who executes the query. It is repeated once for each row in the table.

Query 2-88b

```
SELECT * FROM cust_calls
     WHERE user_id = USER
```

If the user name of the current user is **richc**, Query 2-88b retrieves only those rows in the **cust_calls** table that are owned by that user, as Query Result 2-88 shows.

Query Result 2-88

```
customer_num  110
call_dtime    1998-07-07 10:24
user_id       richc
call_code     L
call_descr    Order placed one month ago (6/7) not received.
res_dtime     1998-07-07 10:30
res_descr     Checked with shipping (Ed Smith). Order sent yesterday- we
              were waiting for goods from ANZ. Next time will call with
              delay if necessary

customer_num  119
call_dtime    1998-07-01 15:00
user_id       richc
call_code     B
call_descr    Bill does not reflect credit from previous order
res_dtime     1998-07-02 08:21
res_descr     Spoke with Jane Akant in Finance. She found the error and is
              sending new bill to customer
```

Using the TODAY Function

The TODAY function returns the current system date. If Query 2-89 is issued when the current system date is July 10, 1998, it returns this one row.

Query 2-89

```
SELECT * FROM orders
    WHERE order_date = TODAY
```

Query Result 2-89

```
order_num      1018
order_date     07/10/1998
customer_num   121
ship_instruct  SW corner of Biltmore Mall
backlog        n
po_num         S22942
ship_date      07/13/1998
ship_weight    70.50
ship_charge    $20.00
paid_date      08/06/1998
```

Using the DBSERVERNAME and SITENAME Functions

You can include the function DBSERVERNAME (or its synonym, SITENAME) in a SELECT statement to find the name of the database server. You can query the DBSERVERNAME for any table that has rows, including system catalog tables.

In Query 2-90, you assign the label **server** to the DBSERVERNAME expression and also select the **tabid** column from the **systables** system catalog table. This table describes database tables, and **tabid** is the serial-interval table identifier.

Query 2-90

```
SELECT DBSERVERNAME server, tabid
    FROM systables
    WHERE tabid <= 4
```

Query Result 2-90

```
server            tabid

montague              1
montague              2
montague              3
montague              4
```

Without the WHERE clause to restrict the values in the **tabid**, the database server name would be repeated for each row of the **systables** table.

Using the HEX Function

In Query 2-91, the HEX function returns the hexadecimal format of three specified columns in the **customer** table.

Query 2-91

```
SELECT HEX (customer_num) hexnum, HEX (zipcode) hexzip,
    HEX (rowid) hexrow
    FROM customer
```

Query Result 2-91

```
hexnum     hexzip     hexrow

0x00000065 0x00016F86 0x00000001
0x00000066 0x00016FA5 0x00000002
0x00000067 0x0001705F 0x00000003
0x00000068 0x00016F4A 0x00000004
0x00000069 0x00016F46 0x00000005
0x0000006A 0x00016F6F 0x00000006
0x0000006B 0x00017060 0x00000007
0x0000006C 0x00016F6F 0x00000008
0x0000006D 0x00016F86 0x00000009
0x0000006E 0x00016F6E 0x0000000A
0x0000006F 0x00016F85 0x0000000B
0x00000070 0x00016F46 0x0000000C
0x00000071 0x00016F49 0x0000000D
0x00000072 0x00016F6E 0x0000000E
0x00000073 0x00016F49 0x0000000F
0x00000074 0x00016F58 0x00000010
0x00000075 0x00016F6F 0x00000011
0x00000076 0x00017191 0x00000012
0x00000077 0x00001F42 0x00000013
0x00000078 0x00014C18 0x00000014
0x00000079 0x00004DBA 0x00000015
0x0000007A 0x0000215C 0x00000016
0x0000007B 0x00007E00 0x00000017
0x0000007C 0x00012116 0x00000018
0x0000007D 0x00000857 0x00000019
0x0000007E 0x0001395B 0x0000001A
0x0000007F 0x0000EBF6 0x0000001B
0x00000080 0x00014C10 0x0000001C
```

Using the DBINFO Function

You can use the DBINFO function in a SELECT statement to find any of the following information:

- The name of a dbspace corresponding to a tablespace number or expression
- The last serial value inserted in a table
- The number of rows processed by selects, inserts, deletes, updates, and execute procedure statements

IDS

- The session ID of the current session
- The name of the host computer on which the database server runs
- The exact version of the database server to which a client application is connected ♦

You can use the DBINFO function anywhere within SQL statements and within stored procedures.

Query 2-92, shows how you might use the DBINFO function to find out the name of the host computer on which the database server runs.

Query 2-92

```
SELECT DBINFO('dbhostname')
    FROM systables
    WHERE tabid = 1
```

Query Result 2-92

```
(constant)

lyceum
```

Without the WHERE clause to restrict the values in the **tabid**, the host name of the computer on which the database server runs would be repeated for each row of the **systables** table.

Query 2-93, shows how you might use the DBINFO function to find out the complete version number and the type of the current database server.

Query 2-93

```
SELECT DBINFO('version','full')
    FROM systables
    WHERE tabid = 1
```

For more information about how to use the DBINFO function to find information about your current database server, database session, or database, see the *Informix Guide to SQL: Syntax*.

IDS

Using the DECODE Function

You can use the DECODE function to convert an expression of one value to another value. The DECODE function has the following form:

```
DECODE(exp_1, exp_2, exp_3, exp_4, exp_5, ..., exp_n, exp_n+1, exp_m )
```

DECODE returns *expr_3* when *exp_2* equals *exp_1*, and returns *exp_5* when *exp_4* equals *exp_1*, and, in general, returns *exp_n+1* when *exp_n* equals *exp_1*.

If several expressions match *exp_1*, DECODE returns *exp_n+1* for the first expression found. If no expression matches *exp_1*, DECODE returns *exp_m*; if no expression matches *exp_1* and there is no *exp_m*, DECODE returns NULL.

Suppose an **employee** table exists that includes **emp_id** and **evaluation** columns. Suppose also that execution of Query 2-96 on the **employee** table returns the rows shown in Query Result 2-96.

Query 2-94

```
SELECT emp_id, evaluation
    FROM employee:
```

Query Result 2-94

```
emp_id              evaluation

012233              great
012344              poor
012677              NULL
012288              good
012555              very good
```

In some cases, you might want to convert a set of values. For example, suppose you want to convert the descriptive values of the **evaluation** column in the preceding example to corresponding numeric values. Query 2-97 shows how you might use the DECODE function to convert values from the **evaluation** column to numeric values for each row in the **employee** table:

Query 2-95

```
SELECT emp_id, DECODE(evaluation, "poor", 0 "fair", 25,
"good", 50,"very good", 75, "great", 100, -1) as evaluation
    FROM employee
```

Query Result 2-95

```
emp_id              evaluation

012233              100
012344              0
012677              -1
012288              50
012555              75
.
```

You can specify any data type for the arguments of the DECODE function provided that the arguments meet the following requirements:

- The arguments *exp_1*, *exp_2*, *exp_4*, ..., *exp_n* all have the same data type or evaluate to a common compatible data type.
- The arguments *exp_3*, *exp_5*,...,*exp_n+1* all have the same data type or evaluate to a common compatible data type.

IDS

Using the NVL Function

You can use the NVL function to convert an expression that evaluates to null to a value that you specify. The NVL function accepts two arguments: the first argument takes the name of the expression to be evaluated; the second argument specifies the value that the function returns when the first argument evaluates to null. If the first argument does not evaluate to null, the function returns the value of the first argument. Suppose an **student** table exists that includes **name** and **address** columns. Suppose also that execution of Query 2-96 on the **student** table returns the rows shown in Query Result 2-96.

Query 2-96

```
SELECT name, address
FROM student:
```

Query Result 2-96

```
name               address

John Smith         333 Vista Drive
Lauren Collier     1129 Greenridge Street
Fred Frith         NULL
Susan Jordan       NULL
```

Query 2-97 includes the NVL function, which returns a new value for each row in the table where the **address** column contains a null value:

Query 2-97

```
SELECT name, NVL(address, "address is unknown") as address
    FROM student
```

Query Result 2-97

```
name                address

John Smith          333 Vista Drive
Lauren Collier      1129 Greenridge Street
Fred Frith          address is unknown
Susan Jordan        address is unknown
```

You can specify any data type for the arguments of the NVL function provided that the two arguments evaluate to a common compatible data type.

If both arguments of the NVL function evaluate to null, the function returns null.

Using Stored Procedures in SELECT Statements

Previous examples in this chapter show SELECT statement expressions that consist of column names, operators, and SQL functions. This section shows expressions that contain a stored procedure call.

Stored procedures contain special Stored Procedure Language (SPL) statements as well as SQL statements. For more information on stored procedures, see Chapter 8, "Creating and Using Stored Procedures."

Stored procedures provide a way to extend the range of functions available; you can perform a subquery on each row you select.

For example, suppose you want a listing of the customer number, the customer's last name, and the number of orders the customer has made. Query 2-98 shows one way to retrieve this information. The **customer** table has **customer_num** and **lname** columns but no record of the number of orders each customer has made. You could write a **get_orders** procedure, which queries the **orders** table for each **customer_num** and returns the number of corresponding orders (labeled **n_orders**).

Query 2-98

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
    FROM customer
```

Query Result 2-98 shows the output from this stored procedure.

Query Result 2-98

```
customer_num lname n_orders

     101 Pauli     1
     102 Sadler    0
     103 Currie    0
     104 Higgins   4
     105 Vector    0
     106 Watson    2
     107 Ream      0
     108 Quinn     0
     109 Miller    0
     110 Jaeger    2
     111 Keyes     1
     112 Lawson    1
     113 Beatty    0
     114 Albertson 0
     115 Grant     1
     116 Parmelee  1
     117 Sipes     2
     118 Baxter    0
     119 Shorter   1
     120 Jewell    1
     121 Wallack   1
     122 O'Brian   1
     123 Hanlon    1
     124 Putnum    1
     125 Henry     0
     126 Neelie    1
     127 Satifer   1
     128 Lessor    0
```

Use stored procedures to encapsulate operations that you frequently perform in your queries. For example, the condition in Query 2-99 contains a procedure, **conv_price**, that converts the unit price of a stock item to a different currency and adds any import tariffs.

Query 2-99

```
SELECT stock_num, manu_code, description
    FROM stock
    WHERE conv_price(unit_price, ex_rate = 1.50,
    tariff = 50.00) < 1000
```

Multiple-Table SELECT Statements

To select data from two or more tables, name these tables in the FROM clause. Add a WHERE clause to create a *join* condition between at least one related column in each table. This WHERE clause creates a temporary composite table in which each pair of rows that satisfies the join condition is linked to form a single row.

A *simple join* combines information from two or more tables based on the relationship between one column in each table. A *composite join* is a join between two or more tables based on the relationship between two or more columns in each table.

To create a join, you must specify a relationship, called a *join condition*, between at least one column from each table. Because the columns are being compared, they must have compatible data types. When you join large tables, performance improves when you index the columns in the join condition.

Data types are described in the *Informix Guide to SQL: Reference* and the *Informix Guide to Database Design and Implementation*. Indexing is discussed in detail in your *Administrator's Guide*.

Creating a Cartesian Product

When you perform a multiple-table query that does not explicitly state a join condition among the tables, you create a *Cartesian product*. A Cartesian product consists of every possible combination of rows from the tables. This result is usually large and unwieldy, and the data is inaccurate.

Query 2-100 selects from two tables and produces a Cartesian product.

Query 2-100

```
SELECT * FROM customer, state
```

Although only 52 rows exist in the **state** table and 28 rows in the **customer** table, the effect of Query 2-100 is to multiply the rows of one table by the rows of the other and retrieve an impractical 1,456 rows, as Query 2-100 shows.

Query Result 2-100

```
customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          AK
sname         Alaska

customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          HI
sname         Hawaii

customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          CA
sname         California
.
.
.
```

Some of the data that is displayed in the concatenated rows is inaccurate. For example, although the **city** and **state** from the **customer** table indicate an address in California, the **code** and **sname** from the **state** table might be for a different state.

Creating a Join

Conceptually, the first stage of any join is the creation of a Cartesian product. To refine or constrain this Cartesian product and eliminate meaningless rows of data, include a WHERE clause with a valid join condition in your SELECT statement.

This section illustrates *equi-joins*, *natural joins*, and *multiple-table joins*. Additional complex forms, such as *self-joins* and *outer joins*, are discussed in Chapter 3, "Composing Advanced SELECT Statements."

Equi-Join

An equi-join is a join based on equality or matching values. This equality is indicated with an equal sign (=) in the comparison operation in the WHERE clause, as Query 2-101 shows.

Query 2-101

```
SELECT * FROM manufact, stock
    WHERE manufact.manu_code = stock.manu_code
```

Query 2-101 joins the **manufact** and **stock** tables on the **manu_code** column. It retrieves only those rows for which the values for the two columns are equal, as Query Result 2-101 shows.

Query Result 2-101

```
manu_code    SMT
manu_name    Smith
lead_time       3
stock_num    1
manu_code    SMT
description  baseball gloves
unit_price   $450.00
unit         case
unit_descr   10 gloves/case

manu_code    SMT
manu_name    Smith
lead_time       3
stock_num    5
manu_code    SMT
description  tennis racquet
unit_price   $25.00
unit         each
unit_descr   each

manu_code    SMT
manu_name    Smith
lead_time       3
stock_num    6
manu_code    SMT
description  tennis ball
unit_price   $36.00
unit         case
unit_descr   24 cans/case

manu_code    ANZ
manu_name    Anza
lead_time       5
stock_num    5
manu_code    ANZ
description  tennis racquet
unit_price   $19.80
unit         each
unit_descr   each
.
.
.
```

In this equi-join, Query Result 2-101 includes the **manu_code** column from both the **manufact** and **stock** tables because the select list requested every column.

You can also create an equi-join with additional constraints, one where the comparison condition is based on the inequality of values in the joined columns. These joins use a relational operator other than the equal sign (=) in the comparison condition that is specified in the WHERE clause.

To join tables that contain columns with the same name, precede each column name with a period and its table name, as Query 2-102 shows.

Query 2-102

```
SELECT order_num, order_date, ship_date, cust_calls.*
    FROM orders, cust_calls
    WHERE call_dtime >= ship_date
        AND cust_calls.customer_num = orders.customer_num
    ORDER BY customer_num
```

Query 2-102 joins the **customer_num** column and then selects only those rows where the **call_dtime** in the **cust_calls** table is greater than or equal to the **ship_date** in the **orders** table. Query Result 2-102 shows the rows that it returns.

Query Result 2-102

```
order_num      1004
order_date     05/22/1998
ship_date      05/30/1998
customer_num   106
call_dtime     1998-06-12 08:20
user_id        maryj
call_code      D
call_descr     Order received okay, but two of the cans of
               ANZ tennis balls within the case were empty
res_dtime      1998-06-12 08:25
res_descr      Authorized credit for two cans to customer,
               issued apology. Called ANZ buyer to report
               the qa problem.

order_num      1008
order_date     06/07/1998
ship_date      07/06/1998
customer_num   110
call_dtime     1998-07-07 10:24
user_id        richc
call_code      L
call_descr     Order placed one month ago (6/7) not received.
res_dtime      1998-07-07 10:30
res_descr      Checked with shipping (Ed Smith). Order out
               yesterday-was waiting for goods from ANZ.
               Next time will call with delay if necessary.

order_num      1023
order_date     07/24/1998
ship_date      07/30/1998
customer_num   127
call_dtime     1998-07-31 14:30
user_id        maryj
call_code      I
call_descr     Received Hero watches (item # 304) instead
               of ANZ watches
res_dtime
res_descr      Sent memo to shipping to send ANZ item 304
               to customer and pickup HRO watches. Should
               be done tomorrow, 8/1
```

Natural Join

A natural join is structured so that the join column does not display data redundantly, as Query 2-103 shows.

Query 2-103

```
SELECT manu_name, lead_time, stock.*
    FROM manufact, stock
    WHERE manufact.manu_code = stock.manu_code
```

Like the example for equi-join, Query 2-103 joins the **manufact** and **stock** tables on the **manu_code** column. Because the select list is more closely defined, the **manu_code** is listed only once for each row retrieved, as Query Result 2-103 shows.

Query Result 2-103

```
manu_name    Smith
lead_time       3
stock_num    1
manu_code    SMT
description  baseball gloves
unit_price   $450.00
unit         case
unit_descr   10 gloves/case

manu_name    Smith
lead_time       3
stock_num    5
manu_code    SMT
description  tennis racquet
unit_price   $25.00
unit         each
unit_descr   each

manu_name    Smith
lead_time       3
stock_num    6
manu_code    SMT
description  tennis ball
unit_price   $36.00
unit         case
unit_descr   24 cans/case

manu_name    Anza
lead_time       5
stock_num    5
manu_code    ANZ
description  tennis racquet
unit_price   $19.80
unit         each
unit_descr   each
.
.
.
```

All joins are *associative*; that is, the order of the joining terms in the WHERE clause does not affect the meaning of the join.

Both of the statements in Query 2-104 create the same natural join.

Query 2-104

```
SELECT catalog.*, description, unit_price, unit, unit_descr
    FROM catalog, stock
    WHERE catalog.stock_num = stock.stock_num
        AND catalog.manu_code = stock.manu_code
        AND catalog_num = 10017

SELECT catalog.*, description, unit_price, unit, unit_descr
    FROM catalog, stock
    WHERE catalog_num = 10017
        AND catalog.manu_code = stock.manu_code
        AND catalog.stock_num = stock.stock_num
```

Each statement retrieves the row that Query Result 2-104 shows.

Query Result 2-104

```
catalog_num  10017
stock_num    101
manu_code    PRC
cat_descr
Reinforced, hand-finished tubular. Polyurethane belted.
Effective against punctures. Mixed tread for super wear
and road grip.
cat_picture  <BYTE value>

cat_advert   Ultimate in Puncture Protection, Tires
             Designed for In-City Riding
description  bicycle tires
unit_price   $88.00
unit         box
unit_descr   4/box
```

Query 2-104 includes a TEXT column, **cat_descr**; a BYTE column, **cat_picture**; and a VARCHAR column, **cat_advert**.

Multiple-Table Join

A multiple-table join connects more than two tables on one or more associated columns; it can be an equi-join or a natural join.

Query 2-105 creates an equi-join on the **catalog**, **stock**, and **manufact** tables and retrieves the following row:

Query 2-105

```
SELECT * FROM catalog, stock, manufact
    WHERE catalog.stock_num = stock.stock_num
        AND stock.manu_code = manufact.manu_code
        AND catalog_num = 10025
```

Query 2-105 retrieves the rows that Query Result 2-105 shows.

Query Result 2-105

```
catalog_num  10025
stock_num    106
manu_code    PRC
cat_descr
Hard anodized alloy with pearl finish; 6mm hex bolt hardware.
Available in lengths of 90-140mm in 10mm increments.
cat_picture  <BYTE value>

cat_advert   ProCycle Stem with Pearl Finish
stock_num    106
manu_code    PRC
description  bicycle stem
unit_price   $23.00
unit         each
unit_descr   each
manu_code    PRC
manu_name    ProCycle
lead_time       9
```

The **manu_code** is repeated three times, once for each table, and **stock_num** is repeated twice.

To avoid the considerable duplication of a multiple-table query such as Query 2-105, include specific columns in the select list to define the SELECT statement more closely, as Query 2-106 shows.

Query 2-106

```
SELECT catalog.*, description, unit_price, unit,
      unit_descr, manu_name, lead_time
    FROM catalog, stock, manufact
    WHERE catalog.stock_num = stock.stock_num
        AND stock.manu_code = manufact.manu_code
        AND catalog_num = 10025
```

Query 2-106 uses a wildcard to select all columns from the table with the most columns and then specifies columns from the other two tables. Query Result 2-106 shows the natural join that Query 2-106 produces. It displays the same information as the previous example, but without duplication.

Query Result 2-106

```
catalog_num  10025
stock_num    106
manu_code    PRC
cat_descr
Hard anodized alloy with pearl finish. 6mm hex bolt hardware.
Available in lengths of 90-140mm in 10mm increments.
cat_picture  <BYTE value>

cat_advert   ProCycle Stem with Pearl Finish
description  bicycle stem
unit_price   $23.00
unit         each
unit_descr   each
manu_name    ProCycle
lead_time       9
```

Some Query Shortcuts

You can use aliases, the INTO TEMP clause, and display labels to speed your way through joins and multiple-table queries and to produce output for other uses.

Using Aliases

You can assign *aliases* to the tables in a SELECT statement to make multiple-table queries shorter and more readable. An alias is a word that immediately follows the name of a table in the FROM clause. You can use it wherever the table name would be used, for instance, as a prefix to the column names in the other clauses.

Query 2-107a

```
SELECT s.stock_num, s.manu_code, s.description,
       s.unit_price, s.unit, c.catalog_num,
       c.cat_descr, c.cat_advert, m.lead_time
    FROM stock s, catalog c, manufact m
    WHERE s.stock_num = c.stock_num
        AND s.manu_code = c.manu_code
        AND s.manu_code = m.manu_code
        AND s.manu_code IN ('HRO', 'HSK')
        AND s.stock_num BETWEEN 100 AND 301
    ORDER BY catalog_num
```

The associative nature of the SELECT statement allows you to use an alias before you define it. In Query 2-107a, the aliases **s** for the **stock** table, **c** for the **catalog** table, and **m** for the **manufact** table are specified in the FROM clause and used throughout the SELECT and WHERE clauses as column prefixes.

Compare the length of Query 2-107a with Query 2-107b, which does not use aliases.

Query 2-107b

```
SELECT stock.stock_num, stock.manu_code, stock.description,
       stock.unit_price, stock.unit, catalog.catalog_num,
       catalog.cat_descr, catalog.cat_advert,
       manufact.lead_time
    FROM stock, catalog, manufact
    WHERE stock.stock_num = catalog.stock_num
        AND stock.manu_code = catalog.manu_code
        AND stock.manu_code = manufact.manu_code
        AND stock.manu_code IN ('HRO', 'HSK')
        AND stock.stock_num BETWEEN 100 AND 301
    ORDER BY catalog_num
```

Query 2-107a and Query 2-107b are equivalent and retrieve the data in Query Result 2-107.

Query Result 2-107

```
stock_num    110
manu_code    HRO
description  helmet
unit_price   $260.00
unit         case
catalog_num  10033
cat_descr
Newest ultralight helmet uses plastic shell. Largest ventilation
channels of any helmet on the market. 8.5 oz.
cat_advert   Lightweight Plastic Slatted with Vents Assures Cool
             Comfort Without Sacrificing Protection
lead_time       4

stock_num    110
manu_code    HSK
description  helmet
unit_price   $308.00
unit         each
catalog_num  10034
cat_descr
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.
cat_advert   Teardrop Design Endorsed by Yellow Jerseys,
             You Can Time the Difference
lead_time       5

stock_num    205
manu_code    HRO
description  3 golf balls
unit_price   $312.00
unit         each
catalog_num  10048
cat_descr
Combination fluorescent yellow and standard white.
cat_advert   HiFlier Golf Balls: Case Includes Fluorescent
             Yellow and Standard White
lead_time       4

stock_num    301
manu_code    HRO
description  running shoes
unit_price   $42.50
unit         each
catalog_num  10050
cat_descr
Engineered for serious training with exceptional stability.
Fabulous shock absorption. Great durability. Specify
mens/womens, size.
cat_advert   Pronators and Supinators Take Heart: A Serious
             Training Shoe For Runners Who Need Motion Control
lead_time       4
```

You cannot use the ORDER BY clause for the TEXT column **cat_descr** or the BYTE column **cat_picture**.

You can also use aliases to shorten your queries on tables that are not in the current database.

Query 2-108 joins columns from two tables that reside in different databases and systems, neither of which is the current database or system.

Query 2-108

```
SELECT order_num, lname, fname, phone
FROM masterdb@central:customer c, sales@western:orders o
    WHERE c.customer_num = o.customer_num
        AND order_num <= 1010
```

By assigning the aliases **c** and **o** to the long *database@system:table* names, **masterdb@central:customer** and **sales@western:orders**, respectively, you can use the aliases to shorten the expression in the WHERE clause and retrieve the data as Query Result 2-108 shows.

Query Result 2-108

```
order_num lname           fname          phone

     1001 Higgins         Anthony        415-368-1100
     1002 Pauli           Ludwig         408-789-8075
     1003 Higgins         Anthony        415-368-1100
     1004 Watson          George         415-389-8789
     1005 Parmelee        Jean           415-534-8822
     1006 Lawson          Margaret       415-887-7235
     1007 Sipes           Arnold         415-245-4578
     1008 Jaeger          Roy            415-743-3611
     1009 Keyes           Frances        408-277-7245
     1010 Grant           Alfred         415-356-1123
```

For more information on how to access tables that are not in the current database, see "Selecting Tables from a Database Other Than the Current Database" on page 2-101 in this manual and the *Informix Guide to SQL: Syntax*.

You also can use *synonyms* as shorthand references to the long names of tables that are not in the current database as well as current tables and views. For details on how to create and use synonyms, see the *Informix Guide to Database Design and Implementation*.

The INTO TEMP Clause

By adding an INTO TEMP clause to your SELECT statement, you can temporarily save the results of a multiple-table query in a separate table that you can query or manipulate without modifying the database. Temporary tables are dropped when you end your SQL session or when your program or report terminates.

Query 2-109 creates a temporary table called **stockman** and stores the results of the query in it. Because all columns in a temporary table must have names, the alias **adj_price** is required.

Query 2-109

```
SELECT DISTINCT stock_num, manu_name, description,
                unit_price, unit_price * 1.05  adj_price
    FROM stock, manufact
    WHERE manufact.manu_code = stock.manu_code
    INTO TEMP stockman
```

Query Result 2-109

```
stock_num manu_name       description     unit_price   adj_price

        1 Hero            baseball gloves    $250.00    $262.5000
        1 Husky           baseball gloves    $800.00    $840.0000
        1 Smith           baseball gloves    $450.00    $472.5000
        2 Hero            baseball           $126.00    $132.3000
        3 Husky           baseball bat       $240.00    $252.0000
        4 Hero            football           $480.00    $504.0000
        4 Husky           football           $960.00   $1008.0000
        .
        .
        .
      306 Shimara         tandem adapter     $190.00    $199.5000
      307 ProCycle        infant jogger      $250.00    $262.5000
      308 ProCycle        twin jogger        $280.00    $294.0000
      309 Hero            ear drops           $40.00     $42.0000
      309 Shimara         ear drops           $40.00     $42.0000
      310 Anza            kick board          $84.00     $88.2000
      310 Shimara         kick board          $80.00     $84.0000
      311 Shimara         water gloves        $48.00     $50.4000
      312 Hero            racer goggles       $72.00     $75.6000
      312 Shimara         racer goggles       $96.00    $100.8000
      313 Anza            swim cap            $60.00     $63.0000
      313 Shimara         swim cap            $72.00     $75.6000
```

You can query on this table and join it with other tables, which avoids a multiple sort and lets you move more quickly through the database. For more information on temporary tables, see your *Administrator's Guide*.

Selecting Tables from a Database Other Than the Current Database

The database that a CONNECT, DATABASE or CREATE DATABASE statement opens is the *current* database. To refer to a table in a database other than the current database, include the database name as part of the table name, as the following SELECT statement illustrates:

```
SELECT name, number FROM salesdb:contacts
```

The database is **salesdb**. The table in **salesdb** is named **contacts**. You can use the same notation in a join. When you must specify the database name explicitly, the long table names can become cumbersome unless you use aliases to shorten them, as the following example shows:

```
SELECT C.custname, S.phone
    FROM salesdb:contacts C, stores:customer S
    WHERE C.custname = S.company
```

You must qualify the database name with a *database server name* to specify a table in a database that a different database server manages. For example, the following SELECT statement refers to table **customer** from database **masterdb**, which resides on the database server **central**:

```
SELECT O.order_num, C.fname, C.lname
    FROM masterdb@central:customer C, sales@boston:orders O
    WHERE C.customer_num = O.Customer_num
    INTO TEMP mycopy
```

In the example, two tables are being joined. The joined rows are stored in a temporary table called **mycopy** in the current database. The tables are located in two database servers, **central** and **boston**.

Informix allows you to *over qualify* table names (to give more information than is required). Because both table names are fully qualified, you cannot tell whether the current database is **masterdb** or **sales**.

Summary

This chapter introduced sample syntax and results for basic kinds of SELECT statements that are used to query on a relational database. The section "Single-Table SELECT Statements" on page 2-11 shows how to perform the following actions:

- Select all columns and rows from a table with the SELECT and FROM clauses
- Select specific columns from a table with the SELECT and FROM clauses
- Select specific rows from a table with the SELECT, FROM, and WHERE clauses
- Use the DISTINCT or UNIQUE keyword in the SELECT clause to eliminate duplicate rows from query results
- Sort retrieved data with the ORDER BY clause and the DESC keyword
- Select and order data that contains non-English characters
- Use the BETWEEN, IN, MATCHES, and LIKE keywords and various relational operators in the WHERE clause to create a comparison condition
- Create comparison conditions that include values, exclude values, find a range of values (with keywords, relational operators, and subscripting), and find a subset of values
- Use exact-text comparisons, variable-length wildcards, and restricted and unrestricted wildcards to perform variable text searches
- Use the logical operators AND, OR, and NOT to connect search conditions or Boolean expressions in a WHERE clause
- Use the ESCAPE keyword to protect special characters in a query
- Search for null values with the IS NULL and IS NOT NULL keywords in the WHERE clause
- Use the FIRST clause to specify that a query returns only a specified number of the rows that match the conditions of the SELECT statement
- Use arithmetic operators in the SELECT clause to perform computations on number fields and display derived data

- Use substrings and subscripting to tailor your queries
- Assign display labels to computed columns as a formatting tool for reports
- Use the aggregate functions in the SELECT clause to calculate and retrieve specific data
- Include the time functions DATE, DAY, MDY, MONTH, WEEKDAY, YEAR, CURRENT, and EXTEND plus the TODAY, LENGTH, and USER functions in your SELECT statements
- Use conversion functions in the SELECT clause to convert between date and character values
- Use string manipulation functions in the SELECT clause to convert between upper and lower case letters or to manipulate character strings in various ways
- Include stored procedures in your SELECT statements

This chapter also introduced simple join conditions that enable you to select and display data from two or more tables. The section "Multiple-Table SELECT Statements" on page 2-88 describes how to perform the following actions:

- Create a Cartesian product
- Include a WHERE clause with a valid join condition in your query to constrain a Cartesian product
- Define and create a natural join and an equi-join
- Join two or more tables on one or more columns
- Use aliases as a shortcut in multiple-table queries
- Retrieve selected data into a separate, temporary table with the INTO TEMP clause to perform computations outside the database

The next chapter explains more complex queries and subqueries; self-joins and outer joins; the GROUP BY and HAVING clauses; and the UNION, INTERSECTION, and DIFFERENCE set operations.

Chapter

3

Composing Advanced SELECT Statements

Using the GROUP BY and HAVING Clauses 3-4
  Using the GROUP BY Clause 3-4
  Using the HAVING Clause 3-8

Creating Advanced Joins 3-10
  Self-Joins 3-11
    Using Rowid Values 3-15
  Outer Joins 3-20
    Simple Join 3-21
    Simple Outer Join on Two Tables 3-23
    Outer Join for a Simple Join to a Third Table 3-25
    Outer Join of Two Tables to a Third Table 3-26
    Joins That Combine Outer Joins 3-28

Subqueries in SELECT Statements 3-30
  Using ALL 3-31
  Using ANY 3-32
  Single-Valued Subqueries 3-33
  Correlated Subqueries 3-35
  Using EXISTS 3-36

Set Operations 3-39
  Union 3-40
  Intersection 3-48
  Difference. 3-50

Summary 3-52

The previous chapter, "Composing Simple SELECT Statements," demonstrates some basic ways to retrieve data from a relational database with the SELECT statement. This chapter increases the scope of what you can do with this powerful SQL statement and enables you to perform more complex database queries and data manipulation.

Whereas the previous chapter focused on five of the clauses in SELECT statement syntax, this chapter adds two more. You can use the GROUP BY clause with aggregate functions to organize rows returned by the FROM clause. You can include a HAVING clause to place conditions on the values that the GROUP BY clause returns.

This chapter also extends the earlier discussion of joins. It illustrates *self-joins*, which enable you to join a table to itself, and four kinds of *outer joins*, in which you apply the keyword OUTER to treat two or more joined tables unequally. It also introduces correlated and uncorrelated subqueries and their operational keywords, shows how to combine queries with the UNION operator, and defines the set operations known as union, intersection, and difference.

Examples in this chapter show how to use some or all of the SELECT statement clauses in your queries. The clauses must appear in the following order:

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY
7. INTO TEMP

For an example of a SELECT statement that uses all these clauses in the correct order, see Query 3-8 on page 3-10.

An additional SELECT statement clause, INTO, which you can use to specify program and host variables in SQL APIs, is described in Chapter 5, "Programming with SQL," as well as in the manuals that come with the product.

Using the GROUP BY and HAVING Clauses

The optional GROUP BY and HAVING clauses add functionality to your SELECT statement. You can include one or both in a basic SELECT statement to increase your ability to manipulate aggregates.

The GROUP BY clause combines similar rows, producing a single result row for each *group* of rows that have the same values for each column listed in the select list. The HAVING clause sets conditions on those groups after you form them. You can use a GROUP BY clause without a HAVING clause, or a HAVING clause without a GROUP BY clause.

Using the GROUP BY Clause

The GROUP BY clause divides a table into sets. This clause is most often combined with aggregate functions that produce summary values for each of those sets. Some examples in Chapter 2, "Composing Simple SELECT Statements" show the use of aggregate functions applied to a whole table. This chapter illustrates aggregate functions applied to groups of rows.

Using the GROUP BY clause without aggregates is much like using the DISTINCT (or UNIQUE) keyword in the SELECT clause. Chapter 2, "Composing Simple SELECT Statements," included the statement found in Query 3-1a.

Query 3-1a

```
SELECT DISTINCT customer_num FROM orders
```

You could also write the statement as Query 3-1b shows.

Query 3-1b

```
SELECT customer_num
    FROM orders
    GROUP BY customer_num
```

Query 3-1a and Query 3-1b return the rows that Query Result 3-1 shows.

Query Result 3-1

```
customer_num

         101
         104
         106
         110
         111
         112
         115
         116
         117
         119
         120
         121
         122
         123
         124
         126
         127
```

The GROUP BY clause collects the rows into sets so that each row in each set has equal customer numbers. With no other columns selected, the result is a list of the unique **customer_num** values.

The power of the GROUP BY clause is more apparent when you use it with aggregate functions.

Query 3-2 retrieves the number of items and the total price of all items for each order.

Query 3-2

```
SELECT order_num, COUNT (*) number, SUM (total_price) price
    FROM items
    GROUP BY order_num
```

The GROUP BY clause causes the rows of the **items** table to be collected into groups, each group composed of rows that have identical **order_num** values (that is, the items of each order are grouped together). After you form the groups, the aggregate functions COUNT and SUM are applied within each group.

Query 3-2 returns one row for each group. It uses labels to give names to the results of the COUNT and SUM expressions, as Query Result 3-2 shows.

Query Result 3-2

```
order_num      number         price

   1001           1         $250.00
   1002           2        $1200.00
   1003           3         $959.00
   1004           4        $1416.00
   1005           4         $562.00
   1006           5         $448.00
   1007           5        $1696.00
   1008           2         $940.00
   .
   .
   .
   1015           1         $450.00
   1016           4         $654.00
   1017           3         $584.00
   1018           5        $1131.00
   1019           1        $1499.97
   1020           2         $438.00
   1021           4        $1614.00
   1022           3         $232.00
   1023           6         $824.00
```

Query Result 3-2 collects the rows of the **items** table into groups that have identical order numbers and computes the COUNT of rows in each group and the sum of the prices.

You cannot include a TEXT or BYTE column in a GROUP BY clause. To *group*, you must be able to *sort*, and no natural sort order exists for TEXT or BYTE data.

Unlike the ORDER BY clause, the GROUP BY clause does not order data. Include an ORDER BY clause *after* your GROUP BY clause if you want to sort data in a particular order or sort on an aggregate in the select list.

Query 3-3 is the same as Query 3-2 but includes an ORDER BY clause to sort the retrieved rows in ascending order of **price**, as Query Result 3-3 shows.

Query 3-3

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
    FROM items
    GROUP BY order_num
    ORDER BY price
```

Query Result 3-3

```
order_num       number          price

   1010            2             $84.00
   1011            1             $99.00
   1013            4            $143.80
   1022            3            $232.00
   1001            1            $250.00
   1020            2            $438.00
   1006            5            $448.00
   1015            1            $450.00
   1009            1            $450.00
  .
  .
  .
   1018            5           $1131.00
   1002            2           $1200.00
   1004            4           $1416.00
   1014            2           $1440.00
   1019            1           $1499.97
   1021            4           $1614.00
   1007            5           $1696.00
```

As stated in Chapter 2, "Composing Simple SELECT Statements," you can use an integer in an ORDER BY clause to indicate the position of a column in the select list. You also can use an integer in a GROUP BY clause to indicate the position of column names or display labels in the group list.

Query 3-4 returns the same rows as Query 3-3, as Query Result 3-3 shows.

Query 3-4

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
    FROM items
    GROUP BY 1
    ORDER BY 3
```

When you build a query, remember that all nonaggregate columns that are in the select list in the SELECT clause must also be included in the group list in the GROUP BY clause. The reason is that a SELECT statement with a GROUP BY clause must return only one row per group. Columns that are listed after GROUP BY are certain to reflect only one distinct value within a group, and that value can be returned. However, a column not listed after GROUP BY might contain different values in the rows that are contained in a group.

As Query 3-5 shows, you can use the GROUP BY clause in a SELECT statement that joins tables.

Query 3-5

```
SELECT o.order_num, SUM (i.total_price)
    FROM orders o, items i
    WHERE o.order_date > '01/01/93'
        AND o.customer_num = 110
        AND o.order_num = i.order_num
    GROUP BY o.order_num
```

Query 3-5 joins the **orders** and **items** tables, assigns table aliases to them, and returns the rows that Query Result 3-5 shows.

Query Result 3-5

```
order_num            (sum)

     1008          $940.00
     1015          $450.00
```

Using the HAVING Clause

To complement a GROUP BY clause, use a HAVING clause to apply one or more qualifying conditions to groups after they are formed. The effect of the HAVING clause on groups is similar to the way the WHERE clause qualifies individual rows. One advantage of using a HAVING clause is that you can include aggregates in the search condition, whereas you cannot include aggregates in the search condition of a WHERE clause.

Each HAVING condition compares one column or aggregate expression of the group with another aggregate expression of the group or with a constant. You can use HAVING to place conditions on both column values and aggregate values in the group list.

Query 3-6 returns the average total price per item on all orders that have more than two items. The HAVING clause tests each group as it is formed and selects those that are composed of more than two rows.

Query 3-6

```
SELECT order_num, COUNT(*) number, AVG (total_price) average
    FROM items
    GROUP BY order_num
    HAVING COUNT(*) > 2
```

Query Result 3-6

```
   order_num      number          average

        1003           3          $319.67
        1004           4          $354.00
        1005           4          $140.50
        1006           5           $89.60
        1007           5          $339.20
        1013           4           $35.95
        1016           4          $163.50
        1017           3          $194.67
        1018           5          $226.20
        1021           4          $403.50
        1022           3           $77.33
        1023           6          $137.33
```

If you use a HAVING clause without a GROUP BY clause, the HAVING condition applies to all rows that satisfy the search condition. In other words, all rows that satisfy the search condition make up a single group.

Query 3-7, a modified version of Query 3-6, returns just one row, the average of all **total_price** values in the table.

Query 3-7

```
SELECT AVG (total_price) average
    FROM items
    HAVING count(*) > 2
```

Query Result 3-7

```
average

$270.97
```

If Query 3-7, like Query 3-6, had included the nonaggregate column **order_ num** in the select list, you would have to include a GROUP BY clause with that column in the group list. In addition, if the condition in the HAVING clause was not satisfied, the output would show the column heading and a message would indicate that no rows were found.

Query 3-8 contains all the SELECT statement clauses that you can use in the Informix version of interactive SQL (the INTO clause that names host variables is available only in an SQL API).

Query 3-8

```
SELECT o.order_num, SUM (i.total_price) price,
      paid_date - order_date span
    FROM orders o, items i
    WHERE o.order_date > '01/01/93'
        AND o.customer_num > 110
        AND o.order_num = i.order_num
    GROUP BY 1, 3
    HAVING COUNT (*) < 5
    ORDER BY 3
    INTO TEMP temptab1
```

Query 3-8 joins the **orders** and **items** tables; employs display labels, table aliases, and integers that are used as column indicators; groups and orders the data; and puts the following results in a temporary table, as Query Result 3-8 shows.

Query Result 3-8

```
order_num              price        span

     1017            $584.00
     1016            $654.00
     1012           $1040.00
     1019           $1499.97          26
     1005            $562.00          28
     1021           $1614.00          30
     1022            $232.00          40
     1010             $84.00          66
     1009            $450.00          68
     1020            $438.00          71
```

Creating Advanced Joins

Chapter 2, "Composing Simple SELECT Statements," shows how to include a WHERE clause in a SELECT statement to join two or more tables on one or more columns. It illustrates natural joins and equi-joins.

This chapter discusses how to use two more complex kinds of joins, self-joins and outer joins. As described for simple joins, you can define aliases for tables and assign display labels to expressions to shorten your multiple-table queries. You can also issue a SELECT statement with an ORDER BY clause that sorts data into a temporary table.

Self-Joins

A join does not always have to involve two different tables. You can join a table to itself, creating a *self-join*. Joining a table to itself can be useful when you want to compare values in a column to other values in the same column.

To create a self-join, list a table twice in the FROM clause, and assign it a different alias each time. Use the aliases to refer to the table in the SELECT and WHERE clauses as if it were two separate tables. (Aliases in SELECT statements are shown in Chapter 2, "Composing Simple SELECT Statements," of this manual and discussed in the *Informix Guide to SQL: Syntax*.)

Just as in joins between tables, you can use arithmetic expressions in self-joins. You can test for null values, and you can use an ORDER BY clause to sort the values in a specified column in ascending or descending order.

Query 3-9 finds pairs of orders where the **ship_weight** differs by a factor of five or more and the **ship_date** is not null. The query then orders the data by **ship_date**.

Query 3-9

```
SELECT x.order_num, x.ship_weight, x.ship_date,
       y.order_num, y.ship_weight, y.ship_date
    FROM orders x, orders y
    WHERE x.ship_weight >= 5 * y.ship_weight
        AND x.ship_date IS NOT NULL
        AND y.ship_date IS NOT NULL
    ORDER BY x.ship_date
```

Query Result 3-9

```
order_num ship_weight ship_date    order_num ship_weight ship_date

     1004       95.80 05/30/1998        1011       10.40 07/03/1998
     1004       95.80 05/30/1998        1020       14.00 07/16/1998
     1004       95.80 05/30/1998        1022       15.00 07/30/1998
     1007      125.90 06/05/1998        1015       20.60 07/16/1998
     1007      125.90 06/05/1998        1020       14.00 07/16/1998
     1007      125.90 06/05/1998        1022       15.00 07/30/1998
     1007      125.90 06/05/1998        1011       10.40 07/03/1998
     1007      125.90 06/05/1998        1001       20.40 06/01/1998
     1007      125.90 06/05/1998        1009       20.40 06/21/1998
     1005       80.80 06/09/1998        1011       10.40 07/03/1998
     1005       80.80 06/09/1998        1020       14.00 07/16/1998
     1005       80.80 06/09/1998        1022       15.00 07/30/1998
     1012       70.80 06/29/1998        1011       10.40 07/03/1998
     1012       70.80 06/29/1998        1020       14.00 07/16/1998
     1013       60.80 07/10/1998        1011       10.40 07/03/1998
     1017       60.00 07/13/1998        1011       10.40 07/03/1998
     1018       70.50 07/13/1998        1011       10.40 07/03/1998
        .
```

If you want to store the results of a self-join into a temporary table, append an INTO TEMP clause to the SELECT statement and assign display labels to at least one set of columns to rename them. Otherwise, the duplicate column names cause an error and the temporary table is not created.

Query 3-10, which is similar to Query 3-9, labels all columns selected from the **orders** table and puts them in a temporary table called **shipping**.

Query 3-10

```
SELECT x.order_num orders1, x.po_num purch1,
      x.ship_date ship1, y.order_num orders2,
      y.po_num purch2, y.ship_date ship2
   FROM orders x, orders y
   WHERE x.ship_weight >= 5 * y.ship_weight
       AND x.ship_date IS NOT NULL
       AND y.ship_date IS NOT NULL
   ORDER BY orders1, orders2
   INTO TEMP shipping
```

If you query with `SELECT *` from that table, you see the rows that Query Result 3-10 shows.

Query Result 3-10

```
orders1 purch1     ship1          orders2 purch2     ship2

   1004 8006       05/30/1998        1011 B77897     07/03/1998
   1004 8006       05/30/1998        1020 W2286      07/16/1998
   1004 8006       05/30/1998        1022 W9925      07/30/1998
   1005 2865       06/09/1998        1011 B77897     07/03/1998
   1005 2865       06/09/1998        1020 W2286      07/16/1998
   1005 2865       06/09/1998        1022 W9925      07/30/1998
   1007 278693     06/05/1998        1001 B77836     06/01/1998
   1007 278693     06/05/1998        1009 4745       06/21/1998
   1007 278693     06/05/1998        1011 B77897     07/03/1998
   1007 278693     06/05/1998        1015 MA003      07/16/1998
   1007 278693     06/05/1998        1020 W2286      07/16/1998
   1007 278693     06/05/1998        1022 W9925      07/30/1998
   1012 278701     06/29/1998        1011 B77897     07/03/1998
   1012 278701     06/29/1998        1020 W2286      07/16/1998
   1013 B77930     07/10/1998        1011 B77897     07/03/1998
   1017 DM354331   07/13/1998        1011 B77897     07/03/1998
   1018 S22942     07/13/1998        1011 B77897     07/03/1998
   1018 S22942     07/13/1998        1020 W2286      07/16/1998
   1019 Z55709     07/16/1998        1011 B77897     07/03/1998
   1019 Z55709     07/16/1998        1020 W2286      07/16/1998
   1019 Z55709     07/16/1998        1022 W9925      07/30/1998
   1023 KF2961     07/30/1998        1011 B77897     07/03/1998
```

You can join a table to itself more than once. The maximum number of self-joins depends on the resources available to you.

The self-join in Query 3-11 creates a list of those items in the **stock** table that are supplied by three manufacturers. The self-join includes the last two conditions in the WHERE clause to eliminate duplicate manufacturer codes in rows that are retrieved.

Query 3-11

```
SELECT s1.manu_code, s2.manu_code, s3.manu_code,
      s1.stock_num, s1.description
    FROM stock s1, stock s2, stock s3
    WHERE s1.stock_num = s2.stock_num
        AND s2.stock_num = s3.stock_num
        AND s1.manu_code < s2.manu_code
        AND s2.manu_code < s3.manu_code
    ORDER BY stock_num
```

Query Result 3-11

```
manu_code manu_code manu_code stock_num description

HRO       HSK       SMT               1 baseball gloves
ANZ       NRG       SMT               5 tennis racquet
ANZ       HRO       HSK             110 helmet
ANZ       HRO       PRC             110 helmet
ANZ       HRO       SHM             110 helmet
ANZ       HSK       PRC             110 helmet
ANZ       HSK       SHM             110 helmet
ANZ       PRC       SHM             110 helmet
HRO       HSK       PRC             110 helmet
HRO       HSK       SHM             110 helmet
HRO       PRC       SHM             110 helmet
HSK       PRC       SHM             110 helmet
ANZ       KAR       NKL             201 golf shoes
ANZ       HRO       NKL             205 3 golf balls
ANZ       HRO       KAR             301 running shoes
 .
 .
 .
HRO       PRC       SHM             301 running shoes
KAR       NKL       PRC             301 running shoes
KAR       NKL       SHM             301 running shoes
KAR       PRC       SHM             301 running shoes
NKL       PRC       SHM             301 running shoes
```

If you want to select rows from a payroll table to determine which employees earn more than their manager, you can construct the self-join that Query 3-12a shows.

Query 3-12a

```
SELECT emp.employee_num, emp.gross_pay, emp.level,
        emp.dept_num, mgr.employee_num, mgr.gross_pay,
        mgr.dept_num, mgr.level
    FROM payroll emp, payroll mgr
    WHERE emp.gross_pay > mgr.gross_pay
        AND emp.level < mgr.level
        AND emp.dept_num = mgr.dept_num
    ORDER BY 4
```

Query 3-12b uses a *correlated subquery* to retrieve and list the 10 highest-priced items ordered.

Query 3-12b

```
SELECT order_num, total_price
    FROM items a
    WHERE 10 >
        (SELECT COUNT (*)
            FROM items b
            WHERE b.total_price < a.total_price)
    ORDER BY total_price
```

Query 3-12b returns the 10 rows that Query Result 3-12 shows.

Query Result 3-12

```
order_num total_price

     1018      $15.00
     1013      $19.80
     1003      $20.00
     1005      $36.00
     1006      $36.00
     1013      $36.00
     1010      $36.00
     1013      $40.00
     1022      $40.00
     1023      $40.00
```

You can create a similar query to find and list the 10 employees in the company who have the most seniority.

Correlated and uncorrelated subqueries are described later in "Subqueries in SELECT Statements" on page 3-30.

Using Rowid Values

The database server assigns a unique rowid to rows in nonfragmented tables. Rows in fragmented tables do not contain the rowid column. For a fragmented table, you can use the WITH ROWIDS clause to add the rowid column to the table.

Informix recommends that you use primary keys as a method of access in your applications rather than rowids. Because primary keys are defined in the ANSI specification of SQL, using them to access data makes your applications more portable. In addition, the database server requires less time to access data in a fragmented table when it uses a primary key than it requires to access the same data when it uses rowid.

AD/XP

Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options does not support rowids for fragmented tables, so you must use column values to identify the rows of a fragmented table. ♦

For more information about rowids and tables, see the *Informix Guide to Database Design and Implementation*.

You can use the hidden *rowid* column in a self-join to locate duplicate values in a table. In the following example, the condition `x.rowid != y.rowid` is equivalent to saying "row *x* is not the same row as row *y*."

Query 3-13 selects data twice from the **cust_calls** table, assigning it the table aliases **x** and **y**.

Query 3-13

```
SELECT x.rowid, x.customer_num
    FROM cust_calls x, cust_calls y
    WHERE x.customer_num = y.customer_num
        AND x.rowid != y.rowid
```

Query 3-13 searches for duplicate values in the **customer_num** column, and for their rowids, finding the pair that Query Result 3-13 shows.

Query Result 3-13

```
rowid customer_num

  515          116
  769          116
```

You can write the last condition as Query 3-13 shows.

```
AND x.rowid != y.rowid

AND NOT x.rowid = y.rowid
```

Another way to locate duplicate values is with a correlated subquery, as Query 3-14 shows.

Query 3-14

```
SELECT x.customer_num, x.call_dtime
    FROM cust_calls x
    WHERE 1 <
        (SELECT COUNT (*) FROM cust_calls y
            WHERE x.customer_num = y.customer_num)
```

Query 3-14 locates the same two duplicate **customer_num** values as Query 3-13 and returns the rows that Query Result 3-14 shows.

Query Result 3-14

```
customer_num call_dtime

         116 1997-11-28 13:34
         116 1997-12-21 11:24
```

You can use the rowid, shown earlier in a self-join, to locate the internal record number that is associated with a row in a database table. The rowid is, in effect, a hidden column in every table. The sequential values of rowid have no special significance and can vary depending on the location of the physical data in the chunk. Your rowid might vary from the example shown. For detailed information about how to use rowids, see your *Administrator's Guide*.

Query 3-15 uses the rowid and the wildcard asterisk symbol (*) in the SELECT clause to retrieve every row in the **manufact** table and their corresponding rowids.

Query 3-15

```
SELECT rowid, * FROM manufact
```

Query Result 3-15

```
rowid manu_code manu_name       lead_time

  257 SMT       Smith               3
  258 ANZ       Anza                5
  259 NRG       Norge               7
  260 HSK       Husky               5
  261 HRO       Hero                4
  262 SHM       Shimara            30
  263 KAR       Karsten            21
  264 NKL       Nikolus             8
  265 PRC       ProCycle            9
```

You also can use the rowid when you select a specific column, as Query 3-16 shows.

Query 3-16

```
SELECT rowid, manu_code FROM manufact
```

Query Result 3-16

```
rowid manu_code

  258 ANZ
  261 HRO
  260 HSK
  263 KAR
  264 NKL
  259 NRG
  265 PRC
  262 SHM
  257 SMT
```

You can use the rowid in the WHERE clause to retrieve rows based on their internal record number. This method is handy when no other unique column exists in a table. Query 3-17 uses a rowid from Query 3-16.

Query 3-17

```
SELECT * FROM manufact WHERE rowid = 263
```

Query 3-17 returns the row that Query Result 3-17 shows.

Query Result 3-17

```
manu_code manu_name       lead_time

KAR       Karsten           21
```

Using the USER Function

To obtain additional information about a table, you can combine the rowid with the USER function.

Query 3-18 assigns the label **username** to the USER expression column and returns this information about the **cust_calls** table.

Query 3-18

```
SELECT USER username, rowid FROM cust_calls
```

Query Result 3-18

```
username           rowid

zenda                257
zenda                258
zenda                259
zenda                513
zenda                514
zenda                515
zenda                769
```

You also can use the USER function in a WHERE clause when you select the rowid.

Query 3-19 returns the rowid for only those rows that are inserted or updated by the user who performs the query.

Query 3-19

```
SELECT rowid FROM cust_calls WHERE user_id = USER
```

For example, if the user **richc** used Query 3-19, the output would be as shown in Query Result 3-19.

Query Result 3-19

```
     rowid

       258
       259
```

Using the DBSERVERNAME Function

You can add the DBSERVERNAME function (or its synonym, SITENAME) to a query to find out where the current database resides.

Query 3-20 finds the database server name and the user name as well as the rowid and the *tabid*, which is the serial-interval table identifier for system catalog tables.

Query 3-20

```
SELECT DBSERVERNAME server, tabid, rowid, USER username
     FROM systables
     WHERE tabid >= 105 OR rowid <= 260
     ORDER BY rowid
```

Query 3-20 assigns display labels to the DBSERVERNAME and USER expressions and returns the 10 rows from the **systables** system catalog table, as Query Result 3-20 shows.

Query Result 3-20

```
server              tabid       rowid username

manatee                 1         257 zenda
manatee                 2         258 zenda
manatee                 3         259 zenda
manatee                 4         260 zenda
manatee               105         274 zenda
manatee               106        1025 zenda
manatee               107        1026 zenda
manatee               108        1027 zenda
manatee               109        1028 zenda
manatee               110        1029 zenda
```

Never store a rowid in a *permanent* table or attempt to use it as a foreign key because the rowid can change. For example, if a table is dropped and then reloaded from external data, all the rowids are different.

The USER and DBSERVERNAME statements are discussed in Chapter 2, "Composing Simple SELECT Statements."

Outer Joins

Chapter 2, "Composing Simple SELECT Statements," shows how to create and use some simple joins. Whereas a simple join treats two or more joined tables equally, an *outer join* treats two or more joined tables *asymmetrically*. An outer join makes one of the tables *dominant* (also called *preserved*) over the other *subservient* tables.

The database server supports the following three basic types of outer joins:

- A simple outer join on two tables
- An outer join for a simple join to a third table
- An outer join of two tables to a third table

This section discusses these types of outer joins. For complete information on the syntax, use, and logic of outer joins, see the *Informix Guide to SQL: Syntax*.

In a *simple join*, the result contains only the combinations of rows from the tables that satisfy the join conditions. *Rows that do not satisfy the join conditions are discarded.*

In an *outer join*, the result contains the combinations of rows from the tables that satisfy the join conditions. In addition, the result preserves rows from the dominant table that would otherwise be discarded because no matching row was found in the subservient table. The dominant-table rows that do not have a matching subservient-table row receive nulls for the columns of the subservient table.

An outer join applies conditions to the subservient table while it sequentially applies the join conditions to the rows of the dominant table. The conditions are expressed in a WHERE clause.

An outer join must have a SELECT clause, a FROM clause, and a WHERE clause. To transform a simple join into an outer join, insert the keyword OUTER directly before the name of the subservient tables in the FROM clause. As shown later in this section, you can include the OUTER keyword more than once in your query.

Before you use outer joins heavily, determine whether one or more simple joins can work. You often can get by with a simple join when you do not need supplemental information from other tables.

The examples in this section use table aliases for brevity. Table aliases are discussed in Chapter 2, "Composing Simple SELECT Statements."

Simple Join

Query 3-21 is an example of the type of simple join on the **customer** and **cust_calls** tables that is shown in Chapter 2, "Composing Simple SELECT Statements."

Query 3-21

```
SELECT c.customer_num, c.lname, c.company,
      c.phone, u.call_dtime, u.call_descr
    FROM customer c, cust_calls u
    WHERE c.customer_num = u.customer_num
```

Query 3-21 returns only those rows in which the customer has made a call to customer service, as Query Result 3-21 shows.

Query Result 3-21

```
customer_num  106
lname         Watson
company       Watson & Son
phone         415-389-8789
call_dtime    1998-06-12 08:20
call_descr    Order was received, but two of the cans of
              ANZ tennis balls within the case were empty

customer_num  110
lname         Jaeger
company       AA Athletics
phone         415-743-3611
call_dtime    1998-07-07 10:24
call_descr    Order placed one month ago (6/7) not received.

customer_num  119
lname         Shorter
company       The Triathletes Club
phone         609-663-6079
call_dtime    1998-07-01 15:00
call_descr    Bill does not reflect credit from previous order

customer_num  121
lname         Wallack
company       City Sports
phone         302-366-7511
call_dtime    1998-07-10 14:05
call_descr    Customer likes our merchandise. Requests that we
              stock more types of infant joggers. Will call back
              to place order.

customer_num  127
lname         Satifer
company       Big Blue Bike Shop
phone         312-944-5691
call_dtime    1998-07-31 14:30
call_descr    Received Hero watches (item # 304) instead of
              ANZ watches

customer_num  116
lname         Parmelee
company       Olympic City
phone         415-534-8822
call_dtime    1997-11-28 13:34
call_descr    Received plain white swim caps (313 ANZ) instead
              of navy with team logo (313 SHM)

customer_num  116
lname         Parmelee
company       Olympic City
phone         415-534-8822
call_dtime    1997-12-21 11:24
call_descr    Second complaint from this customer! Received
              two cases right-handed outfielder gloves (1 HRO)
              instead of one case lefties.
```

Simple Outer Join on Two Tables

Query 3-22 uses the same select list, tables, and comparison condition as the preceding example, but this time it creates a simple outer join.

Query 3-22

```
SELECT c.customer_num, c.lname, c.company,
      c.phone, u.call_dtime, u.call_descr
    FROM customer c, OUTER cust_calls u
    WHERE c.customer_num = u.customer_num
```

The addition of the keyword OUTER before the **cust_calls** table makes it the subservient table. An outer join causes the query to return information on *all* customers, whether or not they have made calls to customer service. All rows from the dominant **customer** table are retrieved, and null values are assigned to columns of the subservient **cust_calls** table, as Query Result 3-22 shows.

Query Result 3-22

```
customer_num  101
lname         Pauli
company       All Sports Supplies
phone         408-789-8075
call_dtime
call_descr

customer_num  102
lname         Sadler
company       Sports Spot
phone         415-822-1289
call_dtime
call_descr

customer_num  103
lname         Currie
company       Phil's Sports
phone         415-328-4543
call_dtime
call_descr

customer_num  104
lname         Higgins
company       Play Ball!
phone         415-368-1100
call_dtime
call_descr

customer_num  105
lname         Vector
company       Los Altos Sports
phone         415-776-3249
call_dtime
call_descr

customer_num  106
lname         Watson
company       Watson & Son
phone         415-389-8789
call_dtime    1998-06-12 08:20
call_descr    Order was received, but two of the cans of
              ANZ tennis balls within the case were empty

customer_num  107
lname         Ream
company       Athletic Supplies
phone         415-356-9876
call_dtime
call_descr

customer_num  108
lname         Quinn
company       Quinn's Sports
phone         415-544-8729
call_dtime
call_descr
    .
```

Outer Join for a Simple Join to a Third Table

Query 3-23 shows an outer join that is the result of a simple join to a third table. This second type of outer join is known as a *nested simple join*.

Query 3-23

```
SELECT c.customer_num, c.lname, o.order_num,
      i.stock_num, i.manu_code, i.quantity
    FROM customer c, OUTER (orders o, items i)
    WHERE c.customer_num = o.customer_num
        AND o.order_num = i.order_num
        AND manu_code IN ('KAR', 'SHM')
    ORDER BY lname
```

Query 3-23 first performs a simple join on the **orders** and **items** tables, retrieving information on all orders for items with a **manu_code** of KAR or SHM. It then performs an outer join to combine this information with data from the dominant **customer** table. An optional ORDER BY clause reorganizes the data into the form that Query Result 3-23 shows.

Query Result 3-23

```
customer_num lname            order_num stock_num manu_code quantity

         114 Albertson
         118 Baxter
         113 Beatty
         103 Currie
         115 Grant
         123 Hanlon                1020       301 KAR              4
         123 Hanlon                1020       204 KAR              2
         125 Henry
         104 Higgins
         110 Jaeger
         120 Jewell                1017       202 KAR              1
         120 Jewell                1017       301 SHM              2
         111 Keyes
         112 Lawson
         128 Lessor
         109 Miller
         126 Neelie
         122 O'Brian               1019       111 SHM              3
         116 Parmelee
         101 Pauli
         124 Putnum                1021       202 KAR              3
         108 Quinn
         107 Ream
         102 Sadler
         127 Satifer               1023       306 SHM              1
         127 Satifer               1023       105 SHM              1
         127 Satifer               1023       110 SHM              1
         119 Shorter               1016       101 SHM              2
         117 Sipes
         105 Vector
         121 Wallack               1018       302 KAR              3
         106 Watson
```

Outer Join of Two Tables to a Third Table

Query 3-24 shows an outer join that is the result of an outer join of each of two tables to a third table. In this third type of outer join, join relationships are possible *only* between the dominant table and the subservient tables.

Query 3-24

```
SELECT c.customer_num, lname, o.order_num,
        order_date, call_dtime
    FROM customer c, OUTER orders o, OUTER cust_calls x
    WHERE c.customer_num = o.customer_num
        AND c.customer_num = x.customer_num
    ORDER BY lname
    INTO TEMP service
```

Query 3-24 individually joins the subservient tables **orders** and **cust_calls** to the dominant **customer** table; it does not join the two subservient tables. An INTO TEMP clause selects the results into a temporary table for further manipulation or queries, as Query Result 3-24 shows.

Query Result 3-24

```
customer_num lname             order_num order_date call_dtime

         114 Albertson
         118 Baxter
         113 Beatty
         103 Currie
         115 Grant                  1010 06/17/1998
         123 Hanlon                 1020 07/11/1998
         125 Henry
         104 Higgins                1003 05/22/1998
         104 Higgins                1001 05/20/1998
         104 Higgins                1013 06/22/1998
         104 Higgins                1011 06/18/1998
         110 Jaeger                 1015 06/27/1998 1998-07-07 10:24
         110 Jaeger                 1008 06/07/1998 1998-07-07 10:24
         120 Jewell                 1017 07/09/1998
         111 Keyes                  1009 06/14/1998
         112 Lawson                 1006 05/30/1998
         109 Miller
         128 Moore
         126 Neelie                 1022 07/24/1998
         122 O'Brian                1019 07/11/1998
         116 Parmelee               1005 05/24/1998 1997-12-21 11:24
         116 Parmelee               1005 05/24/1998 1997-11-28 13:34
         101 Pauli                  1002 05/21/1998
         124 Putnum                 1021 07/23/1998
         108 Quinn
         107 Ream
         102 Sadler
         127 Satifer                1023 07/24/1998 1998-07-31 14:30
         119 Shorter                1016 06/29/1998 1998-07-01 15:00
         117 Sipes                  1007 05/31/1998
         117 Sipes                  1012 06/18/1998
         105 Vector
         121 Wallack                1018 07/10/1998 1998-07-10 14:05
         106 Watson                 1004 05/22/1998 1998-06-12 08:20
         106 Watson                 1014 06/25/1998 1998-06-12 08:20
```

If Query 3-24 had tried to create a join condition between the two subservient tables **o** and **x**, as Query 3-25 shows, an error message would have indicated the creation of a two-sided outer join.

Query 3-25

```
WHERE o.customer_num = x.customer_num
```

Joins That Combine Outer Joins

To achieve multiple levels of nesting, you can create a join that employs any combination of the three types of outer joins. Query 3-26 creates a join that is the result of a combination of a simple outer join on two tables and a second outer join.

Query 3-26

```
SELECT c.customer_num, lname, o.order_num,
      stock_num, manu_code, quantity
    FROM customer c, OUTER (orders o, OUTER items i)
    WHERE c.customer_num = o.customer_num
        AND o.order_num = i.order_num
        AND manu_code IN ('KAR', 'SHM')
    ORDER BY lname
```

Query 3-26 first performs an outer join on the **orders** and **items** tables, retrieving information on all orders for items with a **manu_code** of KAR or SHM. It then performs a second outer join that combines this information with data from the dominant **customer** table. Query 3-26 preserves order numbers that the previous example eliminated, returning rows for orders that do not contain items with either manufacturer code. Query Result 3-26 shows how the ORDER BY clause reorganizes the data.

Query Result 3-26

```
customer_num lname          order_num stock_num manu_code quantity

         114 Albertson
         118 Baxter
         113 Beatty
         103 Currie
         115 Grant               1010
         123 Hanlon              1020       204 KAR              2
         123 Hanlon              1020       301 KAR              4
         125 Henry
         104 Higgins             1011
         104 Higgins             1001
         104 Higgins             1013
         104 Higgins             1003
         110 Jaeger              1008
         110 Jaeger              1015
         120 Jewell              1017       301 SHM              2
         120 Jewell              1017       202 KAR              1
         111 Keyes               1009
         112 Lawson              1006
         128 Lessor
         109 Miller
         126 Neelie              1022
         122 O'Brian             1019       111 SHM              3
         116 Parmelee            1005
         101 Pauli               1002
         124 Putnum              1021       202 KAR              3
         108 Quinn
         107 Ream
         102 Sadler
         127 Satifer             1023       110 SHM              1
         127 Satifer             1023       105 SHM              1
         127 Satifer             1023       306 SHM              1
         119 Shorter             1016       101 SHM              2
         117 Sipes               1012
         117 Sipes               1007
         105 Vector
         121 Wallack             1018       302 KAR              3
         106 Watson              1014
         106 Watson              1004
```

You can state the join conditions in two ways when you apply an outer join to the result of an outer join to a third table. The two subservient tables are joined, but you can join the dominant table to either subservient table without affecting the results if the dominant table and the subservient table share a common column.

Subqueries in SELECT Statements

A SELECT statement *nested* in the WHERE clause of another SELECT statement (or in an INSERT, DELETE, or UPDATE statement) is called a *subquery.* Each subquery must contain a SELECT clause and a FROM clause. A subquery must be enclosed in parentheses so that the database server performs that operation first.

Subqueries can be *correlated* or *uncorrelated.* A subquery (or *inner* SELECT statement) is correlated when the value it produces depends on a value produced by the *outer* SELECT statement that contains it. Any other kind of subquery is considered uncorrelated.

The important feature of a correlated subquery is that, because it depends on a value from the outer SELECT, it must be executed repeatedly, once for every value that the outer SELECT produces. An uncorrelated subquery is executed only once.

You can construct a SELECT statement with a subquery to replace two separate SELECT statements.

Subqueries in SELECT statements allow you to perform the following actions:

- Compare an expression to the result of another SELECT statement
- Determine whether the results of another SELECT statement include an expression
- Determine whether another SELECT statement selects any rows

An optional WHERE clause in a subquery is often used to narrow the search condition.

A subquery selects and returns values to the first or outer SELECT statement. A subquery can return no value, a single value, or a set of values:

- If a subquery returns *no* value, the query does not return any rows. Such a subquery is equivalent to a null value.
- If a subquery returns *one* value, the value is in the form of either one aggregate expression or exactly one row and one column. Such a subquery is equivalent to a single number or character value.
- If a subquery returns a list or *set* of values, the values represent either one row or one column.

The following keywords introduce a subquery in the WHERE clause of a SELECT statement:

- ALL
- ANY
- IN
- EXISTS

You can use any relational operator with ALL and ANY to compare something to every one of (ALL) or to any one of (ANY) the values that the subquery produces. You can use the keyword SOME in place of ANY. The operator IN is equivalent to = ANY. To create the opposite search condition, use the keyword NOT or a different relational operator.

The EXISTS operator tests a subquery to see if it found any values; that is, it asks if the result of the subquery is not null. You cannot use the EXISTS keyword in a subquery that contains a column with a TEXT or BYTE data type.

For the complete syntax that you use to create a condition with a subquery, see the *Informix Guide to SQL: Syntax*.

Using ALL

Use the keyword ALL preceding a subquery to determine whether a comparison is true for every value returned. If the subquery returns no values, the search condition is *true*. (If it returns no values, the condition is true of all the zero values.)

Query 3-27 lists the following information for all orders that contain an item for which the total price is less than the total price on *every* item in order number 1023.

Query 3-27

```
SELECT order_num, stock_num, manu_code, total_price
    FROM items
    WHERE total_price < ALL
        (SELECT total_price FROM items
            WHERE order_num = 1023)
```

Query Result 3-27

```
order_num stock_num manu_code total_price

     1003         9 ANZ            $20.00
     1005         6 SMT            $36.00
     1006         6 SMT            $36.00
     1010         6 SMT            $36.00
     1013         5 ANZ            $19.80
     1013         6 SMT            $36.00
     1018       302 KAR            $15.00
```

Using ANY

Use the keyword ANY (or its synonym SOME) before a subquery to determine whether a comparison is true for at least one of the values returned. If the subquery returns no values, the search condition is *false*. (Because no values exist, the condition cannot be true for one of them.)

Query 3-28 finds the order number of all orders that contain an item for which the total price is greater than the total price of *any one* of the items in order number 1005.

Query 3-28

```
SELECT DISTINCT order_num
    FROM items
    WHERE total_price > ANY
        (SELECT total_price
            FROM items
            WHERE order_num = 1005)
```

Query Result 3-28

```
order_num

     1001
     1002
     1003
     1004
     1005
     1006
     1007
     1008
     1009
     1010
     1011
     1012
     1013
     1014
     1015
     1016
     1017
     1018
     1019
     1020
     1021
     1022
     1023
```

Single-Valued Subqueries

You do not need to include the keyword ALL or ANY if you know the subquery can return *exactly one value* to the outer-level query. A subquery that returns exactly one value can be treated like a function. This kind of subquery often uses an aggregate function because aggregate functions always return single values.

Query 3-29 uses the aggregate function MAX in a subquery to find the **order_num** for orders that include the maximum number of volleyball nets.

Query 3-29

```
SELECT order_num FROM items
    WHERE stock_num = 9
        AND quantity =
            (SELECT MAX (quantity)
                FROM items
                WHERE stock_num = 9)
```

Query Result 3-29

```
order_num

     1012
```

Query 3-30 uses the aggregate function MIN in the subquery to select items for which the total price is higher than 10 times the minimum price.

Query 3-30

```
SELECT order_num, stock_num, manu_code, total_price
    FROM items x
    WHERE total_price >
        (SELECT 10 * MIN (total_price)
            FROM items
            WHERE order_num = x.order_num)
```

Query Result 3-30

```
order_num stock_num manu_code  total_price

     1003         8 ANZ            $840.00
     1018       307 PRC            $500.00
     1018       110 PRC            $236.00
     1018       304 HRO            $280.00
```

Correlated Subqueries

Query 3-31 is an example of a correlated subquery, which returns a list of the 10 latest shipping dates in the **orders** table. It includes an ORDER BY clause after the subquery to order the results because you cannot include ORDER BY within a subquery.

Query 3-31

```
SELECT po_num, ship_date FROM orders main
    WHERE 10 >
        (SELECT COUNT (DISTINCT ship_date)
            FROM orders sub
            WHERE sub.ship_date > main.ship_date)
            AND ship_date IS NOT NULL
    ORDER BY ship_date, po_num
```

The subquery is correlated because the number that it produces depends on **main.ship_date**, a value that the outer SELECT produces. Thus, the subquery must be executed anew for every row that the outer query considers.

Query 3-31 uses the COUNT function to return a value to the main query. The ORDER BY clause then orders the data. The query locates and returns the 16 rows that have the 10 latest shipping dates, as Query Result 3-31 shows.

Query Result 3-31

```
po_num      ship_date

4745        06/21/1998
278701      06/29/1998
429Q        06/29/1998
8052        07/03/1998
B77897      07/03/1998
LZ230       07/06/1998
B77930      07/10/1998
PC6782      07/12/1998
DM354331    07/13/1998
S22942      07/13/1998
MA003       07/16/1998
W2286       07/16/1998
Z55709      07/16/1998
C3288       07/25/1998
KF2961      07/30/1998
W9925       07/30/1998
```

If you use a correlated subquery, such as Query 3-31, on a very large table, you should index the **ship_date** column to improve performance. Otherwise, this SELECT statement is inefficient because it executes the subquery once for every row of the table. For information about indexing and performance issues, see your *Administrator's Guide* and *Performance Guide*.

Using EXISTS

The keyword EXISTS is known as an *existential qualifier* because the subquery is true only if the outer SELECT, as Query 3-32a shows, finds at least one row.

Query 3-32a

```
SELECT UNIQUE manu_name, lead_time
    FROM manufact
    WHERE EXISTS
        (SELECT * FROM stock
            WHERE description MATCHES '*shoe*'
                AND manufact.manu_code = stock.manu_code)
```

You often can construct a query with EXISTS that is equivalent to one that uses IN. Query 3-32b uses an IN predicate to construct a query that returns the same result as Query 3-32a.

Query 3-32b

```
SELECT UNIQUE manu_name, lead_time
    FROM stock, manufact
    WHERE manufact.manu_code IN
        (SELECT manu_code FROM stock
            WHERE description MATCHES '*shoe*')
                AND stock.manu_code = manufact.manu_code
```

Query 3-32a and Query 3-32b return rows for the manufacturers that produce a kind of shoe as well as the lead time for ordering the product. Query Result 3-32 shows the return values.

Query Result 3-32

```
manu_name        lead_time

Anza                 5
Hero                 4
Karsten             21
Nikolus              8
ProCycle             9
Shimara             30
```

Add the keyword NOT to IN or to EXISTS to create a search condition that is the opposite of the condition in the preceding queries. You can also substitute !=ALL for NOT IN.

Query 3-33 shows two ways to do the same thing. One way might allow the database server to do less work than the other, depending on the design of the database and the size of the tables. To find out which query might be better, use the SET EXPLAIN command to get a listing of the query plan. SET EXPLAIN is discussed in your *Performance Guide* and the *Informix Guide to SQL: Syntax*.

Query 3-33

```
SELECT customer_num, company FROM customer
    WHERE customer_num NOT IN
        (SELECT customer_num FROM orders
            WHERE customer.customer_num = orders.customer_num)

SELECT customer_num, company FROM customer
    WHERE NOT EXISTS
        (SELECT * FROM orders
            WHERE customer.customer_num = orders.customer_num)
```

Each statement in Query 3-33 returns the rows that Query Result 3-33 shows, which identify customers who have not placed orders.

Query Result 3-33

```
customer_num company

         102 Sports Spot
         103 Phil's Sports
         105 Los Altos Sports
         107 Athletic Supplies
         108 Quinn's Sports
         109 Sport Stuff
         113 Sportstown
         114 Sporting Place
         118 Blue Ribbon Sports
         125 Total Fitness Sports
         128 Phoenix University
```

The keywords EXISTS and IN are used for the set operation known as *intersection*, and the keywords NOT EXISTS and NOT IN are used for the set operation known as *difference*. These concepts are discussed in "Set Operations" on page 3-39.

Query 3-34 performs a subquery on the **items** table to identify all the items in the **stock** table that have not yet been ordered.

Query 3-34

```
SELECT stock.* FROM stock
    WHERE NOT EXISTS
        (SELECT * FROM items
            WHERE stock.stock_num = items.stock_num
                AND stock.manu_code = items.manu_code)
```

Query 3-34 returns the rows that Query Result 3-34 shows.

Query Result 3-34

```
stock_num manu_code description unit_price unit unit_descr


      101  PRC        bicycle tires       $88.00  box   4/box
      102  SHM        bicycle brakes     $220.00  case  4 sets/case
      102  PRC        bicycle brakes     $480.00  case  4 sets/case
      105  PRC        bicycle wheels      $53.00  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  SHM        pedal binding      $200.00  case  4 pairs/case
      110  ANZ        helmet             $244.00  case  4/case
      110  HRO        helmet             $260.00  case  4/case
      112  SHM        12-spd, assmbld    $549.00  each  each
      113  SHM        18-spd, assmbld    $685.90  each  each
      201  KAR        golf shoes          $90.00  each  each
      202  NKL        metal woods        $174.00  case  2 sets/case
      203  NKL        irons/wedge        $670.00  case  2 sets/case
      205  NKL        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  PRC        running shoes       $75.00  each  each
      301  ANZ        running shoes       $95.00  each  each
      302  HRO        ice pack             $4.50  each  each
      303  KAR        socks               $36.00  box   24 pairs/box
      305  HRO        first-aid kit       $48.00  case  4/case
      306  PRC        tandem adapter     $160.00  each  each
      308  PRC        twin jogger        $280.00  each  each
      309  SHM        ear drops           $40.00  case  20/case
      310  SHM        kick board          $80.00  case  10/case
      310  ANZ        kick board          $84.00  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
```

No logical limit exists to the number of subqueries a SELECT statement can have, but the size of any statement is physically limited when it is considered as a character string. However, this limit is probably larger than any practical statement that you are likely to compose.

Perhaps you want to check whether information has been entered correctly in the database. One way to find errors in a database is to write a query that returns output only when errors exist. A subquery of this type serves as a kind of *audit query*, as Query 3-35 shows.

Query 3-35

```
SELECT * FROM items
    WHERE total_price != quantity *
        (SELECT unit_price FROM stock
            WHERE stock.stock_num = items.stock_num
                AND stock.manu_code = items.manu_code)
```

Query 3-35 returns only those rows for which the total price of an item on an order is not equal to the stock unit price times the order quantity. If no discount has been applied, such rows were probably entered incorrectly in the database. The query returns rows only when errors occur. If information is correctly inserted into the database, no rows are returned.

Query Result 3-35

```
item_num order_num stock_num manu_code quantity total_price

       1      1004         1 HRO              1     $960.00
       2      1006         5 NRG              5     $190.00
```

Set Operations

The standard set operations *union*, *intersection*, and *difference* let you manipulate database information. These three operations let you use SELECT statements to check the integrity of your database after you perform an update, insert, or delete. They can be useful when you transfer data to a history table, for example, and want to verify that the correct data is in the history table before you delete the data from the original table.

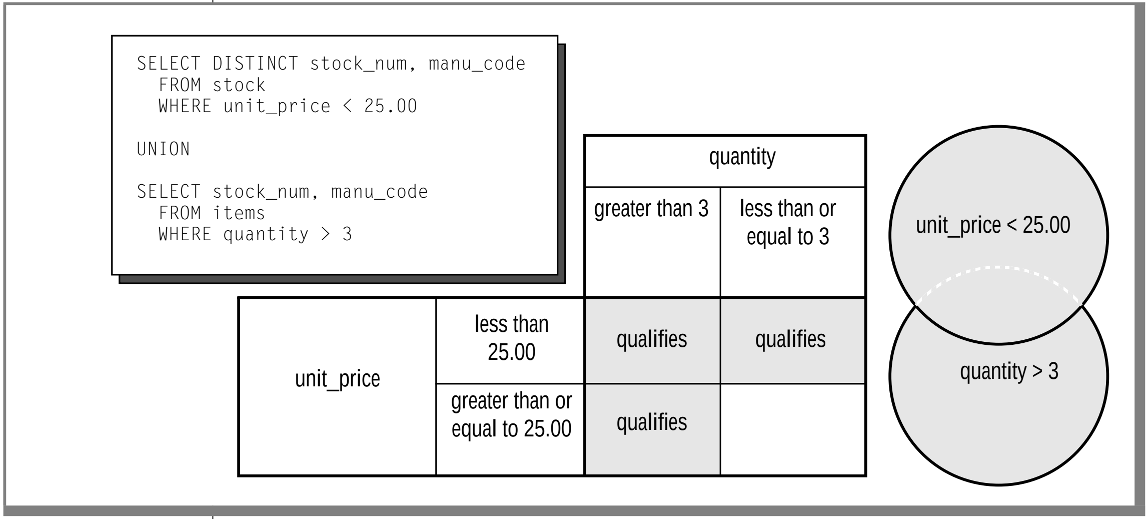
Union

The union operation uses the UNION keyword, or *operator*, to combine two queries into a single *compound query*. You can use the UNION operator between two or more SELECT statements to *unite* them and produce a temporary table that contains rows that exist in any or all of the original tables. You can also use the UNION operator in the definition of a view.

You cannot use a UNION operator inside a subquery.

Figure 3-1 illustrates the union set operation.

Figure 3-1
The Union Set Operation



```
SELECT DISTINCT stock_num, manu_code
  FROM stock
  WHERE unit_price < 25.00

UNION

SELECT stock_num, manu_code
  FROM items
  WHERE quantity > 3
```

		quantity	
		greater than 3	less than or equal to 3
unit_price	less than 25.00	qualifies	qualifies
	greater than or equal to 25.00	qualifies	

The UNION keyword selects all rows from the two queries, removes duplicates, and returns what is left. Because the results of the queries are combined into a single result, the select list in each query must have the same number of columns. Also, the corresponding columns that are selected from each table must contain the same data type (CHARACTER data type columns must be the same length), and these corresponding columns must all allow or all disallow nulls.

Query 3-36 performs a union on the **stock_num** and **manu_code** columns in the **stock** and **items** tables.

Query 3-36

```
SELECT DISTINCT stock_num, manu_code
    FROM stock
    WHERE unit_price < 25.00

UNION

SELECT stock_num, manu_code
    FROM items
    WHERE quantity > 3
```

Query 3-36 selects those items that have a unit price of less than $25.00 or that have been ordered in quantities greater than three and lists their **stock_num** and **manu_code**, as Query Result 3-36 shows.

Query Result 3-36

```
stock_num manu_code

        5 ANZ
        5 NRG
        5 SMT
        9 ANZ
      103 PRC
      106 PRC
      201 NKL
      301 KAR
      302 HRO
      302 KAR
```

If you include an ORDER BY clause, it must follow Query 3-36 and use an integer, not an identifier, to refer to the ordering column. Ordering takes place after the set operation is complete.

Query 3-37

```
SELECT DISTINCT stock_num, manu_code
    FROM stock
    WHERE unit_price < 25.00

UNION

SELECT stock_num, manu_code
    FROM items
    WHERE quantity > 3
    ORDER BY 2
```

The compound query in Query 3-37 selects the same rows as Query 3-36 but displays them in order of the manufacturer code, as Query Result 3-37 shows.

Query Result 3-37

```
stock_num manu_code

        5 ANZ
        9 ANZ
      302 HRO
      301 KAR
      302 KAR
      201 NKL
        5 NRG
      103 PRC
      106 PRC
        5 SMT
```

By default, the UNION keyword excludes duplicate rows. Add the optional keyword ALL, as Query 3-38 shows, to retain the duplicate values.

Query 3-38

```
SELECT stock_num, manu_code
    FROM stock
    WHERE unit_price < 25.00

UNION ALL

SELECT stock_num, manu_code
    FROM items
    WHERE quantity > 3
    ORDER BY 2
    INTO TEMP stockitem
```

Query 3-38 uses the UNION ALL keywords to unite two SELECT statements and adds an INTO TEMP clause after the final SELECT to put the results into a temporary table. It returns the same rows as Query 3-37 but also includes duplicate values.

Query Result 3-38

```
stock_num manu_code

        9 ANZ
        5 ANZ
        9 ANZ
        5 ANZ
        9 ANZ
        5 ANZ
        5 ANZ
        5 ANZ
      302 HRO
      302 KAR
      301 KAR
      201 NKL
        5 NRG
        5 NRG
      103 PRC
      106 PRC
        5 SMT
        5 SMT
```

Corresponding columns in the select lists for the combined queries must have identical data types, but the columns do not need to use the same identifier.

Query 3-39 selects the **state** column from the **customer** table and the corresponding **code** column from the **state** table.

Query 3-39

```
SELECT DISTINCT state
    FROM customer
    WHERE customer_num BETWEEN 120 AND 125

UNION

SELECT DISTINCT code
    FROM state
    WHERE sname MATCHES '*a'
```

Query Result 3-39 returns state code abbreviations for customer numbers 120 through 125 and for states whose **sname** ends in a.

Query Result 3-39

```
state

AK
AL
AZ
CA
DE
FL
GA
IA
IN
LA
MA
MN
MT
NC
ND
NE
NJ
NV
OK
PA
SC
SD
VA
WV
```

In compound queries, the column names or display labels in the first SELECT statement are the ones that appear in the results. Thus, in Query 3-39, the column name **state** from the first SELECT statement is used instead of the column name **code** from the second.

Query 3-40 performs a union on three tables. The maximum number of unions depends on the practicality of the application and any memory limitations.

Query 3-40

```
SELECT stock_num, manu_code
    FROM stock
    WHERE unit_price > 600.00

UNION ALL

SELECT stock_num, manu_code
    FROM catalog
    WHERE catalog_num = 10025

UNION ALL

SELECT stock_num, manu_code
    FROM items
    WHERE quantity = 10
    ORDER BY 2
```

Query 3-40 selects items where the **unit_price** in the **stock** table is greater than $600, the **catalog_num** in the **catalog** table is `10025`, or the **quantity** in the **items** table is `10`; and the query orders the data by **manu_code**. Query Result 3-40 shows the return values.

Query Result 3-40

```
stock_num manu_code

        5 ANZ
        9 ANZ
        8 ANZ
        4 HSK
        1 HSK
      203 NKL
        5 NRG
      106 PRC
      113 SHM
```

For the complete syntax of the SELECT statement and the UNION operator, see the *Informix Guide to SQL: Syntax*. For information specific to the INFORMIX-ESQL/C product and any limitations that involve the INTO clause and compound queries, see the *INFORMIX-ESQL/C Programmer's Manual*.

Query 3-41 uses a combined query to select data into a temporary table and then adds a simple query to order and display it. You must separate the combined and simple queries with a semicolon.

The combined query uses a literal in the select list to tag the output of part of a union so it can be distinguished later. The tag is given the label **sortkey**. The simple query uses that tag as a sort key to order the retrieved rows.

Query 3-41

```
SELECT '1' sortkey, lname, fname, company,
      city, state, phone
    FROM customer x
    WHERE state = 'CA'

UNION

SELECT '2' sortkey, lname, fname, company,
      city, state, phone
    FROM customer y
    WHERE state <> 'CA'
    INTO TEMP calcust;

SELECT * FROM calcust
    ORDER BY 1
```

Query 3-41 creates a list in which the most frequently called customers, those from California, appear first, as Query Result 3-41 shows.

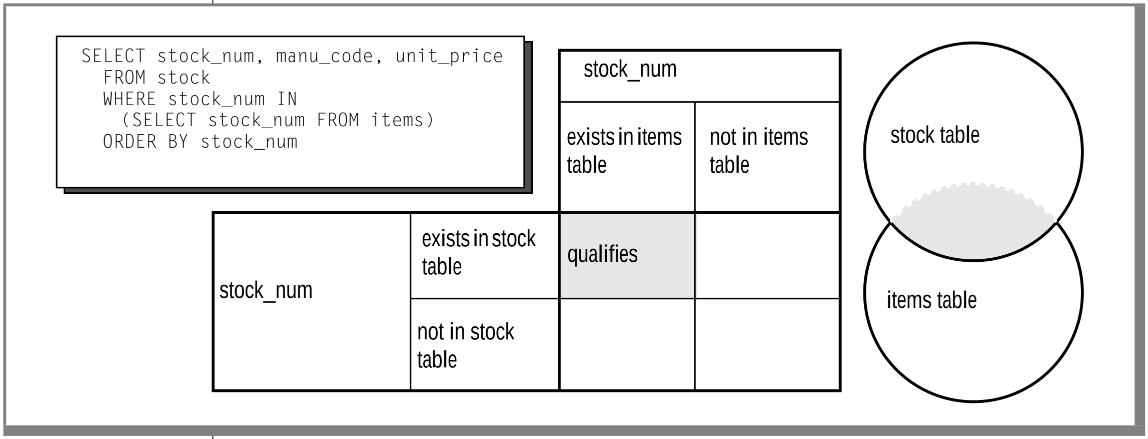
Query Result 3-41

```
sortkey  1
lname    Baxter
fname    Dick
company  Blue Ribbon Sports
city     Oakland
state    CA
phone    415-655-0011

sortkey  1
lname    Beatty
fname    Lana
company  Sportstown
city     Menlo Park
state    CA
phone    415-356-9982

sortkey  1
lname    Currie
fname    Philip
company  Phil’s Sports
city     Palo Alto
state    CA
phone    415-328-4543

sortkey  1
lname    Grant
fname    Alfred
company  Gold Medal Sports
city     Menlo Park
state    CA
phone    415-356-1123
.
.
.
sortkey  2
lname    Satifer
fname    Kim
company  Big Blue Bike Shop
city     Blue Island
state    NY
phone    312-944-5691

sortkey  2
lname    Shorter
fname    Bob
company  The Triathletes Club
city     Cherry Hill
state    NJ
phone    609-663-6079

sortkey  2
lname    Wallack
fname    Jason
company  City Sports
city     Wilmington
state    DE
phone    302-366-7511
```

Intersection

The *intersection* of two sets of rows produces a table containing rows that exist in both the original tables. Use the keyword EXISTS or IN to introduce subqueries that show the intersection of two sets. Figure 3-2 illustrates the intersection set operation.

Figure 3-2
The Intersection Set Operation



Query 3-42 is an example of a nested SELECT statement that shows the intersection of the **stock** and **items** tables.

Query 3-42

```
SELECT stock_num, manu_code, unit_price
    FROM stock
    WHERE stock_num IN
        (SELECT stock_num FROM items)
    ORDER BY stock_num
```

Query Result 3-42 contains all the elements from both sets, returning the following 57 rows.

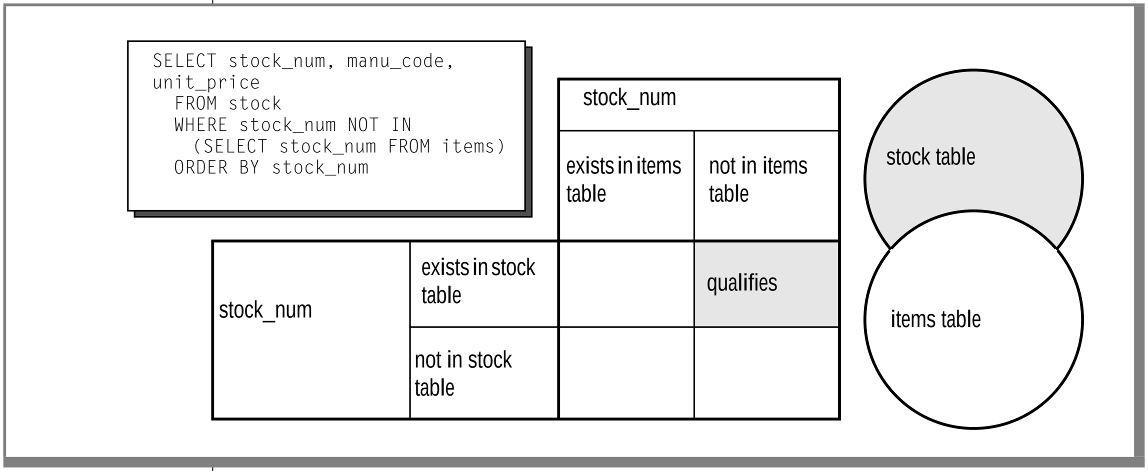
Query Result 3-42

```
stock_num manu_code unit_price

        1 HRO          $250.00z
        1 HSK          $800.00
        1 SMT          $450.00
        2 HRO          $126.00
        3 HSK          $240.00
        3 SHM          $280.00
        4 HRO          $480.00
        4 HSK          $960.00
        5 ANZ           $19.80
        5 NRG           $28.00
        5 SMT           $25.00
        6 ANZ           $48.00
        6 SMT           $36.00
        7 HRO          $600.00
        8 ANZ          $840.00
        9 ANZ           $20.00
      101 PRC           $88.00
      101 SHM           $68.00
      103 PRC           $20.00
      104 PRC           $58.00
      105 PRC           $53.00
      105 SHM           $80.00
      109 PRC           $30.00
      109 SHM          $200.00
      110 ANZ          $244.00
      110 HRO          $260.00
      110 HSK          $308.00
      110 PRC          $236.00
      110 SHM          $228.00
      111 SHM          $499.99
      114 PRC          $120.00
      201 ANZ           $75.00
      201 KAR           $90.00
      201 NKL           $37.50
      202 KAR          $230.00
      202 NKL          $174.00
      204 KAR           $45.00
      205 ANZ          $312.00
      205 HRO          $312.00
      205 NKL          $312.00
      301 ANZ           $95.00
      301 HRO           $42.50
      301 KAR           $87.00
      301 NKL           $97.00
      301 PRC           $75.00
      301 SHM          $102.00
      302 HRO            $4.50
      302 KAR            $5.00
      303 KAR           $36.00
      303 PRC           $48.00
      304 ANZ          $170.00
      304 HRO          $280.00
      306 PRC          $160.00
      306 SHM          $190.00
      307 PRC          $250.00
      309 HRO           $40.00
      309 SHM           $40.00
```

Difference

The *difference* between two sets of rows produces a table that contains rows in the first set that are not also in the second set. Use the keywords NOT EXISTS or NOT IN to introduce subqueries that show the difference between two sets. Figure 3-3 illustrates the difference set operation.

Figure 3-3
The Difference Set Operation



```
SELECT stock_num, manu_code,
unit_price
  FROM stock
  WHERE stock_num NOT IN
    (SELECT stock_num FROM items)
  ORDER BY stock_num
```

Query 3-43 is an example of a nested SELECT statement that shows the difference between the **stock** and **items** tables.

Query 3-43

```
SELECT stock_num, manu_code, unit_price
    FROM stock
    WHERE stock_num NOT IN
        (SELECT stock_num FROM items)
    ORDER BY stock_num
```

Query Result 3-43 contains all the elements from only the first set, which returns 17 rows.

Query Result 3-43

```
stock_num manu_code unit_price

      102 PRC          $480.00
      102 SHM          $220.00
      106 PRC           $23.00
      107 PRC           $70.00
      108 SHM           $45.00
      112 SHM          $549.00
      113 SHM          $685.90
      203 NKL          $670.00
      305 HRO           $48.00
      308 PRC          $280.00
      310 ANZ           $84.00
      310 SHM           $80.00
      311 SHM           $48.00
      312 HRO           $72.00
      312 SHM           $96.00
      313 ANZ           $60.00
      313 SHM           $72.00
```

Summary

This chapter builds on concepts introduced in Chapter 2, “Composing Simple SELECT Statements.” It provides sample syntax and results for more advanced kinds of SELECT statements, which are used to query a relational database. This chapter presents the following material:

- Introduces the GROUP BY and HAVING clauses, which can be used with aggregates to return groups of rows and apply conditions to those groups
- Describes how to use the rowid to retrieve internal record numbers from tables and system-catalog tables and discusses the internal table identifier or tabid
- Shows how to join a table to itself with a self-join to compare values in a column with other values in the same column and to identify duplicates
- Introduces the keyword OUTER, explains how an outer join treats two or more tables asymmetrically, and provides examples of the four kinds of outer join
- Describes how to nest a SELECT statement in the WHERE clause of another SELECT statement to create correlated and uncorrelated subqueries and shows how to use aggregate functions in subqueries
- Demonstrates how to use the keywords ALL, ANY, EXISTS, IN, and SOME to create subqueries, and the effect of adding the keyword NOT or a relational operator
- Discusses the union, intersection, and difference set operations
- Shows how to use the UNION and UNION ALL keywords to create compound queries that consist of two or more SELECT statements

Chapter

Modifying Data

4

Statements That Modify Data 4-3

Deleting Rows 4-4
Deleting All Rows of a Table 4-4
Deleting a Known Number of Rows 4-5
Deleting an Unknown Number of Rows 4-5
Complicated Delete Conditions 4-6

Inserting Rows 4-7
Single Rows 4-7
Possible Column Values 4-8
Listing Specific Column Names 4-9
Multiple Rows and Expressions 4-10
Restrictions on the Insert Selection 4-11

Updating Rows 4-12
Selecting Rows to Update 4-13
Updating with Uniform Values 4-14
Restrictions on Updates 4-15
Updating with Selected Values 4-15
Using a CASE Expression to Update a Column 4-16
Using a Join to Update a Column 4-17

Privileges on a Database 4-17
Database-Level Privileges 4-18
Table-Level Privileges 4-18
Displaying Table Privileges 4-19

Data Integrity. 4-20
Entity Integrity 4-21
Semantic Integrity 4-21
Referential Integrity 4-22
Using the ON DELETE CASCADE Option 4-24

Example of Cascading Deletes 4-25
Restrictions on Cascading Deletes 4-26
Object Modes and Violation Detection 4-26
Object Modes for Constraints. 4-26
Object Modes for Unique Indexes 4-27
Object Modes for Triggers 4-28
SQL Statements and Examples 4-28

Interrupted Modifications 4-28
Transactions 4-30
Transaction Logging 4-30
Transaction Logging for Informix Dynamic Server with AD/XP Options 4-31
Logging and Cascading Deletes 4-31
Specifying Transactions 4-32

Backups and Logs with Informix Database Servers 4-33

Concurrency and Locks 4-34

Data Replication 4-35
Informix Database Server Data Replication 4-36

Summary 4-37

Modifying data is fundamentally different from querying data. Querying data involves examining the contents of tables. Modifying data involves *changing* the contents of tables.

Think about what happens if the system hardware or software fails during a query. In this case, the effect on the application can be severe, but the database itself is unharmed. However, if the system fails while a modification is under way, the state of the database is in doubt. Obviously, a database in an uncertain state has far-reaching implications. Before you delete, insert, or update rows in a database, ask yourself the following questions:

- Is user access to the database and its tables secure; that is, are specific users given limited database and table-level privileges?
- Does the modified data preserve the existing integrity of the database?
- Are systems in place that make the database relatively immune to external events that might cause system or hardware failures?

If you cannot answer yes to each of these questions, do not panic. Solutions to all these problems are built into the Informix database servers. After a description of the statements that modify data, this chapter discusses these solutions. The *Informix Guide to Database Design and Implementation* covers these topics in greater detail.

Statements That Modify Data

The following statements modify data:

- DELETE
- INSERT
- UPDATE

Although these SQL statements are relatively simple when compared with the more advanced SELECT statements, use them carefully because they change the contents of the database.

Deleting Rows

The DELETE statement removes any row or combination of rows from a table. You cannot recover a deleted row after the transaction is committed. (Transactions are discussed under "Interrupted Modifications" on page 4-28. For now, think of a transaction and a statement as the same thing.)

When you delete a row, you must also be careful to delete any rows of other tables whose values depend on the deleted row. If your database enforces referential constraints, you can use the ON DELETE CASCADE option of the CREATE TABLE or ALTER TABLE statements to allow deletes to cascade from one table in a relationship to another. For more information on referential constraints and the ON DELETE CASCADE option, refer to "Referential Integrity" on page 4-22.

Deleting All Rows of a Table

The DELETE statement specifies a table and usually contains a WHERE clause that designates the row or rows that are to be removed from the table. If the WHERE clause is left out, all rows are deleted. *Do not execute the following statement*:

```
DELETE FROM customer
```

Because this DELETE statement does not contain a WHERE clause, all rows from the **customer** table are deleted. If you attempt an unconditional delete using the DB-Access or Relational Object Manager menu options, the program warns you and asks for confirmation. However, an unconditional delete from within a program can occur without warning.

Deleting a Known Number of Rows

The WHERE clause in a DELETE statement has the same form as the WHERE clause in a SELECT Statement. You can use it to designate exactly which row or rows should be deleted. You can delete a customer with a specific customer number, as the following example shows:

```
DELETE FROM customer WHERE customer_num = 175
```

In this example, because the **customer_num** column has a unique constraint, you can ensure that no more than one row is deleted.

Deleting an Unknown Number of Rows

You can also choose rows that are based on nonindexed columns, as the following example shows:

```
DELETE FROM customer WHERE company = 'Druid Cyclery'
```

Because the column that is tested does not have a unique constraint, this statement might delete more than one row. (Druid Cyclery might have two stores, both with the same name but different customer numbers.)

To find out how many rows a DELETE statement affects, select the count of qualifying rows from the **customer** table for Druid Cyclery.

```
SELECT COUNT(*) FROM customer WHERE company = 'Druid Cyclery'
```

You can also select the rows and display them to ensure that they are the ones you want to delete.

Using a SELECT statement as a test is only an approximation, however, when the database is available to multiple users concurrently. Between the time you execute the SELECT statement and the subsequent DELETE statement, other users could have modified the table and changed the result. In this example, another user might perform the following actions:

- Insert a new row for another customer named Druid Cyclery
- Delete one or more of the Druid Cyclery rows before you insert the new row
- Update a Druid Cyclery row to have a new company name, or update some other customer to have the name Druid Cyclery

Although it is not likely that other users would do these things in that brief interval, the possibility does exist. This same problem affects the UPDATE statement. Ways of addressing this problem are discussed under "Concurrency and Locks" on page 4-34, and in greater detail in Chapter 7, "Programming for a Multiuser Environment."

Another problem you might encounter is a hardware or software failure before the statement finishes. In this case, the database might have deleted no rows, some rows, or all specified rows. The *state* of the database is unknown, which is undesirable. To prevent this situation, use transaction logging, as discussed in "Interrupted Modifications" on page 4-28.

Complicated Delete Conditions

The WHERE clause in a DELETE statement can be almost as complicated as the one in a SELECT statement. It can contain multiple conditions that are connected by AND and OR, and it might contain subqueries.

Suppose you discover that some rows of the **stock** table contain incorrect manufacturer codes. Rather than update them, you want to delete them so that they can be reentered. You know that these rows, unlike the correct ones, have no matching rows in the **manufact** table. The fact that these incorrect rows have no matching rows in the **manufact** table allows you to write a DELETE statement such as the one in the following example:

```
DELETE FROM stock
    WHERE 0 = (SELECT COUNT(*) FROM manufact
         WHERE manufact.manu_code = stock.manu_code)
```

The subquery counts the number of rows of **manufact** that match; the count is 1 for a correct row of **stock** and 0 for an incorrect one. The latter rows are chosen for deletion.

One way to develop a DELETE statement with a complicated condition is to first develop a SELECT statement that returns precisely the rows to be deleted. Write it as `SELECT *`; when it returns the desired set of rows, change `SELECT *` to read `DELETE` and execute it once more.

The WHERE clause of a DELETE statement cannot use a subquery that tests the same table. That is, when you delete from **stock**, you cannot use a subquery in the WHERE clause that also selects from **stock**.

The key to this rule is in the FROM clause. If a table is named in the FROM clause of a DELETE statement, it cannot also appear in the FROM clause of a subquery of the DELETE statement.

Inserting Rows

The INSERT statement adds a new row, or rows, to a table. The statement has two basic functions. It can create a single new row using column values you supply, or it can create a group of new rows using data selected from other tables.

Single Rows

In its simplest form, the INSERT statement creates one new row from a list of column values and puts that row in the table. The following statement shows how to add a row to the **stock** table:

```
INSERT INTO stock
    VALUES(115, 'PRC', 'tire pump', 108, 'box', '6/box')
```

The **stock** table has the following columns:

- **stock_num** (a number that identifies the type of merchandise)
- **manu_code** (a foreign key to the **manufact** table)
- **description** (a description of the merchandise)
- **unit_price** (the unit price of the merchandise)
- **unit** (of measure)
- **unit_descr** (characterizes the unit of measure)

The values that are listed in the VALUES clause in the preceding example have a one-to-one correspondence with the columns of the **stock** table. To write a VALUES clause, you must know the columns of the tables as well as their sequence from first to last.

Possible Column Values

The VALUES clause accepts *only* constant values, *not* expressions. You can supply the following values:

- Literal numbers
- Literal datetime values
- Literal interval values
- Quoted strings of characters
- The word NULL for a null value
- The word TODAY for the current date
- The word CURRENT for the current date and time
- The word USER for your user name
- The word DBSERVERNAME (or SITENAME) for the name of the computer where the database server is running

Some columns of a table might not allow null values. If you attempt to insert NULL in such a column, the statement is rejected. Or a column in the table might not permit duplicate values. If you specify a value that is a duplicate of one that is already in such a column, the statement is rejected. Some columns might even *restrict* the possible column values allowed. You use data integrity constraints to restrict columns. For more information on data integrity constraints, see “Data Integrity” on page 4-20.

Only one column in a table can have the SERIAL data type. The database server generates values for a serial column. To make this happen when you insert values, specify the value zero for the serial column. The database server generates the next actual value in sequence. Serial columns do not allow null values.

You can specify a nonzero value for a serial column (as long as it does not duplicate any existing value in that column), and the database server uses the value. However, that nonzero value might set a new starting point for values that the database server generates. The next value the database server generates for you is one greater than the maximum value in the column.

Do not specify the currency symbols for columns that contain money values. Just specify the numeric value of the amount.

The database server can convert between numeric and character data types. You can give a string of numeric characters (for example, `'-0075.6'`) as the value of a numeric column. The database server converts the numeric string to a number. An error occurs only if the string does not represent a number.

You can specify a number or a date as the value for a character column. The database server converts that value to a character string. For example, if you specify TODAY as the value for a character column, a character string that represents the current date is used. (The **DBDATE** environment variable specifies the format that is used.)

Listing Specific Column Names

You do not have to specify values for every column. Instead, you can list the column names after the table name and then supply values for only those columns that you named. The following example shows a statement that inserts a new row into the **stock** table:

```
INSERT INTO stock (stock_num,description,unit_price,manu_code)
   VALUES (115,'tyre pump',114,'SHM')
```

Only the data for the stock number, description, unit price, and manufacturer code is provided. The database server supplies the following values for the remaining columns:

- It generates a serial number for an unlisted serial column.
- It generates a default value for a column with a specific default associated with it.
- It generates a null value for any column that allows nulls but it does not specify a default value for any column that specifies null as the default value.

  This means that you must list and supply values for all columns that do not specify a default value or do not permit nulls.

You can list the columns in any order, as long as the values for those columns are listed in the same order. For information about how to designate null or default values for a column, see the *Informix Guide to Database Design and Implementation*.

After the INSERT statement in the preceding example is executed, the following new row is inserted into the **stock** table:

```
stock_num  manu_code description  unit_price  unit  unit_descr

115        SHM        tyre pump   114
```

Both **unit** and **unit_descr** are blank, which indicates that null values exist in those two columns. Because the **unit** column permits nulls, the number of tire pumps that were purchased for $114 is not known. Of course, if a default value of `box` were specified for this column, then `box` would be the unit of measure. In any case, when you insert values into specific columns of a table, pay attention to what data is needed for that row.

Multiple Rows and Expressions

The other major form of the INSERT statement replaces the VALUES clause with a SELECT statement. This feature allows you to insert the following data:

- Multiple rows with only one statement (each time the SELECT statement returns a row, a row is inserted)
- Calculated values (the VALUES clause permits only constants) because the select list can contain expressions

For example, suppose a follow-up call is required for every order that has been paid for but not shipped. The INSERT statement in the following example finds those orders and inserts a row in **cust_calls** for each order:

```
INSERT INTO cust_calls (customer_num, call_descr)
    SELECT customer_num, order_num FROM orders
        WHERE paid_date IS NOT NULL
        AND ship_date IS NULL
```

This SELECT statement returns two columns. The data from these columns (in each selected row) is inserted into the named columns of the **cust_calls** table. Then, an order number (from **order_num**, a serial column) is inserted into the call description, which is a character column. Remember that the database server allows you to insert integer values into a character column. It automatically converts the serial number to a character string of decimal digits.

Restrictions on the Insert Selection

The following list contains the restrictions on the SELECT statement for inserting rows:

- It cannot contain an INTO clause.
- It cannot contain an INTO TEMP clause.
- It cannot contain an ORDER BY clause.
- It cannot refer to the table into which you are inserting rows.

AD/XP

With Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, a SELECT statement can contain an ORDER BY clause in an INSERT SELECT statement. ♦

The INTO, INTO TEMP, and ORDER BY clause restrictions are minor. The INTO clause is not useful in this context. (It is discussed in Chapter 5, "Programming with SQL.") To work around the INTO TEMP clause restriction, first select the data you want to insert into a temporary table and then insert the data from the temporary table with the INSERT statement. Likewise, the lack of an ORDER BY clause is not important. If you need to ensure that the new rows are physically ordered in the table, you can first select them into a temporary table and order it, and then insert from the temporary table. You can also apply a physical order to the table using a clustered index after all insertions are done.

The last restriction is more serious because it prevents you from naming the same table in both the INTO clause of the INSERT statement and the FROM clause of the SELECT statement. Naming the same table in both the INTO clause of the INSERT statement and the FROM clause of the SELECT statement causes the database server to enter an endless loop in which each inserted row is reselected and reinserted.

In some cases, however, you might want to select from the same table into which you must insert data. For example, suppose that you have learned that the Nikolus company supplies the same products as the Anza company, but at half the price. You want to add rows to the **stock** table to reflect the difference between the two companies. Optimally, you want to select data from all the Anza stock rows and reinsert it with the Nikolus manufacturer code. However, you cannot select from the same table into which you are inserting.

To get around this restriction, select the data you want to insert into a temporary table. Then select from that temporary table in the INSERT statement as the following example shows:

```
SELECT stock_num, 'NIK' temp_manu, description, unit_price/2
        half_price, unit, unit_descr FROM stock
    WHERE manu_code = 'ANZ'
        AND stock_num < 110
    INTO TEMP anzrows;

INSERT INTO stock SELECT * FROM anzrows;

DROP TABLE anzrows;
```

This SELECT statement takes existing rows from **stock** and substitutes a literal value for the manufacturer code and a computed value for the unit price. These rows are then saved in a temporary table, **anzrows**, which is immediately inserted into the **stock** table.

When you insert multiple rows, a risk exists that one of the rows contains invalid data that might cause the database server to report an error. When such an error occurs, the statement terminates early. Even if no error occurs, a very small risk exists that a hardware or software failure might occur while the statement is executing (for example, the disk might fill up).

In either event, you cannot easily tell how many new rows were inserted. If you repeat the statement in its entirety, you might create duplicate rows, or you might not. Because the database is in an unknown state, you cannot know what to do. The answer lies in using transactions, as discussed in "Interrupted Modifications" on page 4-28.

Updating Rows

You use the UPDATE statement to change the contents of one or more columns in one or more existing rows of a table. This statement takes two fundamentally different forms. One lets you assign specific values to columns by name; the other lets you assign a list of values (that might be returned by a SELECT statement) to a list of columns. In either case, if you are updating rows, and some of the columns have data integrity constraints, the data you change must be within the constraints placed on those columns. For more information on data integrity constraints, refer to "Data Integrity" on page 4-20.

Selecting Rows to Update

Either form of the UPDATE statement can end with a WHERE clause that determines which rows are modified. If you omit the WHERE clause, all rows are modified. The WHERE clause can be quite complicated to select the precise set of rows that need changing. The only restriction on the WHERE clause is that the table that you update cannot be named in the FROM clause of a subquery.

The first form of an UPDATE statement uses a series of assignment clauses to specify new column values, as the following example shows:

```
UPDATE customer
    SET fname = 'Barnaby', lname = 'Dorfler'
    WHERE customer_num = 103
```

The WHERE clause selects the row you want to update. In the demonstration database, the **customer.customer_num** column is the primary key for that table, so this statement can update no more than one row.

You can also use subqueries in the WHERE clause. Suppose that the Anza Corporation issues a safety recall of their tennis balls. As a result, any unshipped orders that include stock number 6 from manufacturer ANZ must be put on back order, as the following example shows:

```
UPDATE orders
    SET backlog = 'y'
    WHERE ship_date IS NULL
    AND order_num IN
        (SELECT DISTINCT items.order_num FROM items
            WHERE items.stock_num = 6
            AND items.manu_code = 'ANZ')
```

This subquery returns a column of order numbers (zero or more). The UPDATE operation then tests each row of **orders** against the list and performs the update if that row matches.

Updating with Uniform Values

Each assignment after the keyword SET specifies a new value for a column. That value is applied uniformly to every row that you update. In the examples in the previous section, the new values were constants, but you can assign any expression, including one based on the column value itself. Suppose the manufacturer code HRO has raised all prices by 5 percent, and you must update the **stock** table to reflect this increase. Use a statement such as the following:

```
UPDATE stock
    SET unit_price = unit_price * 1.05
    WHERE manu_code = 'HRO'
```

You can also use a subquery as part of the assigned value. When a subquery is used as an element of an expression, it must return exactly one value (one column and one row). Perhaps you decide that for any stock number, you must charge a higher price than any manufacturer of that product. You need to update the prices of all unshipped orders. The SELECT statements in the following example specify the criteria:

```
UPDATE items
    SET total_price = quantity *
        (SELECT MAX (unit_price) FROM stock
            WHERE stock.stock_num = items.stock_num)
    WHERE items.order_num IN
        (SELECT order_num FROM orders
            WHERE ship_date IS NULL)
```

The first SELECT statement returns a single value: the highest price in the **stock** table for a particular product. The first SELECT statement is a correlated subquery because, when a value from **items** appears in the WHERE clause for the first SELECT statement, you must execute it for every row that you update.

The second SELECT statement produces a list of the order numbers of unshipped orders. It is an uncorrelated subquery that is executed once.

Restrictions on Updates

Restrictions exist on the use of subqueries when you modify data. In particular, you cannot query the table that is being modified. You *can* refer to the present value of a column in an expression, as in the example that increments the **unit_price** column by 5 percent. You can also refer to a value of a column in a WHERE clause in a subquery, as in the example that updated the **stock** table, in which the **items** table is updated and **items.stock_num** is used in a join expression.

AD/XP

With Dynamic Server with AD and XP Options, you cannot use a subquery in the SET clause of an UPDATE statement. ♦

The need to update and query a table at the same time does not occur often in a well-designed database. (For complete information about database design, see the *Informix Guide to Database Design and Implementation*.) However, you might want to update and query at the same time when a database is first being developed, before its design has been carefully thought through. A typical problem arises when a table inadvertently and incorrectly contains a few rows with duplicate values in a column that should be unique. You might want to delete the duplicate rows or update only the duplicate rows. Either way, a test for duplicate rows inevitably requires a subquery on the same table that you want to modify, which is not allowed in an UPDATE statement or DELETE statement. Chapter 6, “Modifying Data Through SQL Programs,” discusses how to use an *update cursor* to perform this kind of modification.

Updating with Selected Values

The second form of UPDATE statement replaces the list of assignments with a single bulk assignment, in which a list of columns is set equal to a list of values. When the values are simple constants, this form is nothing more than the form of the previous example with its parts rearranged, as the following example shows:

```
UPDATE customer
    SET (fname, lname) = ('Barnaby', 'Dorfler')
    WHERE customer_num = 103
```

No advantage exists to writing the statement this way. In fact, it is harder to read because it is not obvious which values are assigned to which columns.

However, when the values to be assigned come from a single SELECT statement, this form makes sense. Suppose that changes of address are to be applied to several customers. Instead of updating the **customer** table each time a change is reported, the new addresses are collected in a single temporary table named **newaddr**. It contains columns for the customer number and the address-related fields of the **customer** table. Now the time comes to apply all the new addresses at once.

```
UPDATE customer
    SET (address1, address2, city, state, zipcode) =
        ((SELECT address1, address2, city, state, zipcode
            FROM newaddr
            WHERE newaddr.customer_num=customer.customer_num))
    WHERE customer_num IN
        (SELECT customer_num FROM newaddr)
```

The values for multiple columns are produced by a single SELECT statement. If you rewrite this example in the other form, with an assignment for each updated column, you must write five SELECT statements, one for each column to be updated. Not only is such a statement harder to write but it also takes much longer to execute.

Tip: *In SQL API programs, you can use record or host variables to update values. For more information, refer to Chapter 5, "Programming with SQL."*

Using a CASE Expression to Update a Column

The CASE expression allows a statement to return one of several possible results, depending on which of several condition tests evaluates to TRUE.

The following example shows how to use a CASE statement in an UPDATE statement to increase the unit price of certain items in the **stock** table:

```
UPDATE stock
    SET unit_price = CASE
                        WHEN stock_num = 1
                         AND manu_code = "HRO"
                        THEN unit_price = unit_price * 1.2
                        WHEN stock_num = 1
                         AND manu_code = "SMT"
                        THEN unit_price = unit_price * 1.1
                        ELSE 0
                     END
```

You must include at least one WHEN clause within the CASE expression; subsequent WHEN clauses and the ELSE clause are optional. If no WHEN condition evaluates to true, the resulting value is null.

AD/XP

Using a Join to Update a Column

Dynamic Server with AD and XP Options allows you to use a join on tables to determine which columns to update. You can use columns from any table that you list in the FROM clause in the SET clause to specify values for the columns and rows to update.

When you use the FROM clause, you must include the name of the table in which the update is to be performed. Otherwise, an error results. The following example illustrates how you can use the UPDATE statement with a FROM clause:

```
UPDATE t SET a = t2.a FROM t, t2 WHERE t.b = t2.b
```

In the preceding example, the statement performs the same action as it does when you omit the FROM clause altogether. You are allowed to specify more than one table in the FROM clause of the UPDATE statement. However, if you specify only one table, it must be the target table.

Privileges on a Database

You can use the following database privileges to control who accesses a database:

- Database-level privileges
- Table-level privileges
- Column-level privileges
- Procedure-level privileges

This section briefly describes the database- and table-level privileges. For complete information about database privileges, see the *Informix Guide to Database Design and Implementation*. For a list of privileges and a description of the GRANT and REVOKE statements, see the *Informix Guide to SQL: Syntax*.

Database-Level Privileges

When you create a database, you are the only one who can access it until you, as the owner or database administrator (DBA) of the database, grant database-level privileges to others. The following table shows the database-level privileges.

Privilege	**Purpose**
Connect	Allows you to open a database, issue queries, and create and place indexes on temporary tables.
Resource	Allows you to create permanent tables.
DBA	Allows you to perform several additional functions as the DBA.

Table-Level Privileges

When you create a table in a database that is not ANSI compliant, all users have access privileges to the table until you, as the owner of the table, revoke table-level privileges from specific users. The following table introduces the four privileges that govern how users can access a table.

Privilege	**Purpose**
Select	Granted on a table-by-table basis and allows you to select rows from a table. (This privilege can be limited to specific columns in a table.)
Delete	Allows you to delete rows.
Insert	Allows you to insert rows.
Update	Allows you to update existing rows (that is, to change their content).

The people who create databases and tables often grant the Connect and Select privileges to **public** so that all users have them. If you can query a table, you have at least the Connect and Select privileges for that database and table.

You need the other table-level privileges to modify data. The owners of tables often withhold these privileges or grant them only to specific users. As a result, you might not be able to modify some tables that you can query freely.

Because these privileges are granted on a table-by-table basis, you can have only Insert privileges on one table and only Update privileges on another, for example. The Update privileges can be restricted even further to specific columns in a table.

For more information on these and other table-level privileges, see the *Informix Guide to Database Design and Implementation*.

Displaying Table Privileges

If you are the owner of a table (that is, if you created it), you have all privileges on that table. Otherwise, you can determine the privileges you have for a certain table by querying the system catalog. The system catalog consists of system tables that describe the database structure. The privileges granted on each table are recorded in the **systabauth** system table. To display these privileges, you must also know the unique identifier number of the table. This number is specified in the **systables** system table. To display privileges granted on the **orders** table, you might enter the following SELECT statement:

```
SELECT * FROM systabauth
    WHERE tabid = (SELECT tabid FROM systables
                        WHERE tabname = 'orders')
```

The output of the query resembles the following example.

```
grantorgranteetabidtabauth

tfecitmutator101su-i-x--
tfecitprocrustes101s--idx--
tfecitpublic101s--i-x--
```

The grantor is the user who *grants* the privilege. The grantor is usually the owner of the table but can be another user empowered by the grantor. The grantee is the user to whom the privilege is granted, and the grantee **public** means “any user with Connect privilege.” If your user name does not appear, you have only those privileges granted to **public**.

The **tabauth** column specifies the privileges granted. The letters in each row of this column are the initial letters of the privilege names except that `i` means Insert and `x` means Index. In this example, **public** has Select, Insert, and Index privileges. Only the user **mutator** has Update privileges, and only the user **procrustes** has Delete privileges.

Before the database server performs any action for you (for example, execution of a DELETE statement), it performs a query similar to the preceding one. If you are not the owner of the table, and if the database server cannot find the necessary privilege on the table for your user name or for **public**, it refuses to perform the operation.

Data Integrity

The INSERT, UPDATE, and DELETE statements modify data in an existing database. Whenever you modify existing data, the *integrity* of the data can be affected. For example, an order for a nonexistent product could be entered into the **orders** table, a customer with outstanding orders could be deleted from the **customer** table, or the order number could be updated in the **orders** table and *not* in the **items** table. In each of these cases, the integrity of the stored data is lost.

Data integrity is actually made up of the following parts:

- Entity integrity

  Each row of a table has a unique identifier.
- Semantic integrity

  The data in the columns properly reflects the types of information the column was designed to hold.
- Referential integrity

  The relationships between tables are enforced.

Well-designed databases incorporate these principles so that when you modify data, the database itself prevents you from doing anything that might harm the integrity of the data.

Entity Integrity

An entity is any person, place, or thing to be recorded in a database. Each table represents an entity, and each row of a table represents an instance of that entity. For example, if *order* is an entity, the **orders** table represents the idea of an order and *each row* in the table represents a specific order.

To identify each row in a table, the table must have a primary key. The primary key is a unique value that identifies each row. This requirement is called the *entity integrity constraint*.

For example, the **orders** table primary key is **order_num**. The **order_num** column holds a unique system-generated order number for each row in the table. To access a row of data in the **orders** table, you can use the following SELECT statement:

```
SELECT * FROM orders WHERE order_num = 1001
```

Using the order number in the WHERE clause of this statement enables you to access a row easily because the order number uniquely identifies that row. If the table allowed duplicate order numbers, it would be almost impossible to access one single row, because all other columns of this table allow duplicate values.

For more information on primary keys and entity integrity, see the *Informix Guide to Database Design and Implementation*.

Semantic Integrity

Semantic integrity ensures that data entered into a row reflects an allowable value for that row. The value must be within the *domain*, or allowable set of values, for that column. For example, the **quantity** column of the **items** table permits only numbers. If a value outside the domain can be entered into a column, the semantic integrity of the data is violated.

The following constraints enforce semantic integrity:

- Data type

  The data type defines the types of values that you can store in a column. For example, the data type SMALLINT allows you to enter values from -32,767 to 32,767 into a column.
- Default value

  The default value is the value inserted into the column when an explicit value is not specified. For example, the **user_id** column of the **cust_calls** table defaults to the login name of the user if no name is entered.
- Check constraint

  The check constraint specifies conditions on data inserted into a column. Each row inserted into a table must meet these conditions. For example, the **quantity** column of the **items** table might check for quantities greater than or equal to one.

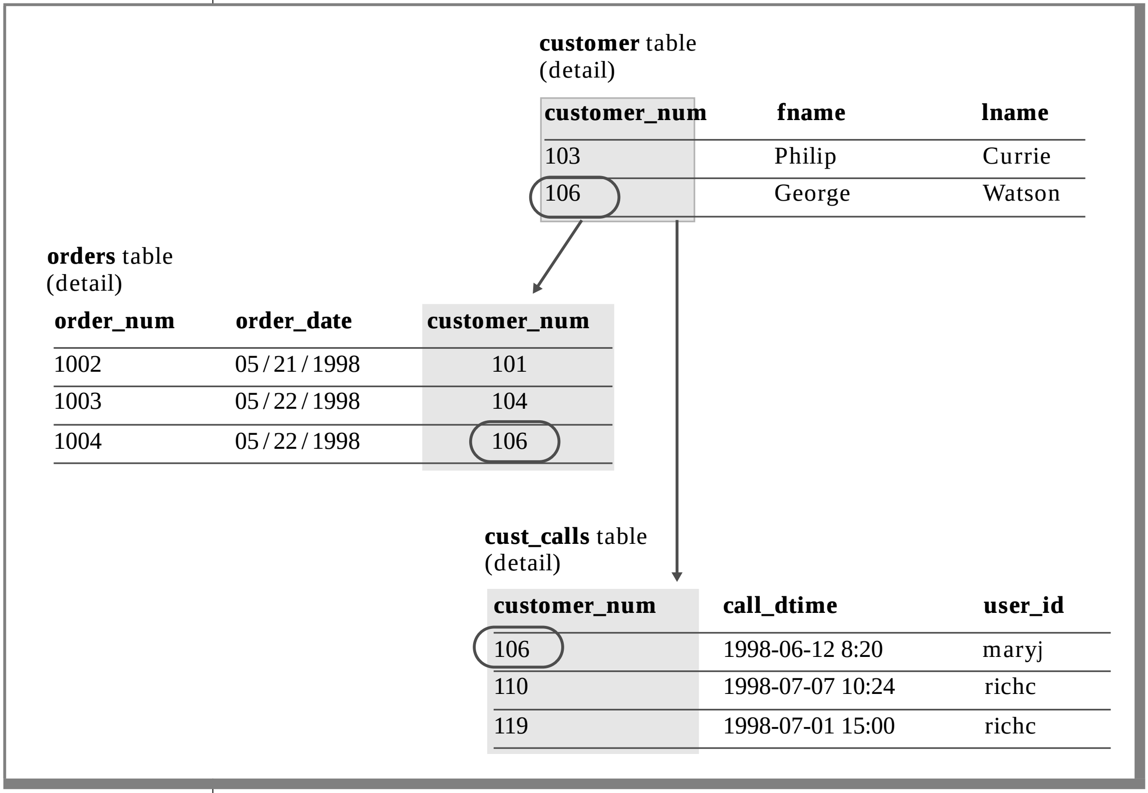
  For more information on how to use semantic integrity constraints in database design, see the *Informix Guide to Database Design and Implementation*.

Referential Integrity

Referential integrity refers to the relationship *between* tables. Because each table in a database must have a primary key, this primary key can appear in other tables because of its relationship to data within those tables. When a primary key from one table appears in another table, it is called a foreign key.

Foreign keys *join* tables and establish dependencies between tables. Tables can form a hierarchy of dependencies in such a way that if you change or delete a row in one table, you destroy the meaning of rows in other tables. For example, Figure 4-1 shows that the **customer_num** column of the **customer** table is a primary key for that table and a foreign key in the **orders** and **cust_call** tables. Customer number 106, George Watson, is *referenced* in both the **orders** and **cust_calls** tables. If customer 106 is deleted from the **customer** table, the link between the three tables and this particular customer is destroyed.

Figure 4-1
Referential Integrity in the Demonstration Database



customer table
(detail)

customer_num	fname	lname
103	Philip	Currie
106	George	Watson

orders table
(detail)

order_num	order_date	customer_num
1002	05/21/1998	101
1003	05/22/1998	104
1004	05/22/1998	106

cust_calls table
(detail)

customer_num	call_dtime	user_id
106	1998-06-12 8:20	maryj
110	1998-07-07 10:24	richc
119	1998-07-01 15:00	richc

When you delete a row that contains a primary key or update it with a different primary key, you destroy the meaning of any rows that contain that value as a foreign key. Referential integrity is the logical dependency of a foreign key on a primary key. The *integrity* of a row that contains a foreign key depends on the integrity of the row that it *references*—the row that contains the matching primary key.

By default, the database server does not allow you to violate referential integrity and gives you an error message if you attempt to delete rows from the parent table before you delete rows from the child table. You can, however, use the ON DELETE CASCADE option to cause deletes from a parent table to trip deletes on child tables. See "Using the ON DELETE CASCADE Option" on page 4-24.

To define primary and foreign keys, and the relationship between them, use the CREATE TABLE and ALTER TABLE statements. For more information on these statements, see the *Informix Guide to SQL: Syntax*. For information about how to build a data model with primary and foreign keys, see the *Informix Guide to Database Design and Implementation*.

Using the ON DELETE CASCADE Option

To maintain referential integrity when you delete rows from a primary key for a table, use the ON DELETE CASCADE option in the REFERENCES clause of the CREATE TABLE and ALTER TABLE statements. This option allows you to delete a row from a parent table and its corresponding rows in matching child tables with a single delete command.

Locking During Cascading Deletes

During deletes, locks are held on all qualifying rows of the parent and child tables. When you specify a delete, the delete that is requested from the parent table occurs before any referential actions are performed.

What Happens to Multiple Children Tables

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

Logging Must Be Turned On

You must turn logging on in your current database for cascading deletes to work. Logging and cascading deletes are discussed in "Transaction Logging" on page 4-30.

Example of Cascading Deletes

Suppose you have two tables with referential integrity rules applied, a parent table, **accounts**, and a child table, **sub_accounts**. The following CREATE TABLE statements define the referential constraints:

```
CREATE TABLE accounts (
 acc_num SERIAL primary key,
 acc_type INT,
 acc_descr CHAR(20));

CREATE TABLE sub_accounts (
 sub_acc INTEGER primary key,
 ref_num INTEGER REFERENCES accounts (acc_num) ON DELETE CASCADE,
 sub_descr CHAR(20));
```

The primary key of the **accounts** table, the **acc_num** column, uses a SERIAL data type, and the foreign key of the **sub_accounts** table, the **ref_num** column, uses an INTEGER data type. Combining the SERIAL data type on the primary key and the INTEGER data type on the foreign key is allowed. Only in this condition can you mix and match data types. The SERIAL data type is an INTEGER, and the database automatically generates the values for the column. All other primary and foreign key combinations must match explicitly. For example, a primary key that is defined as CHAR must match a foreign key that is defined as CHAR.

The definition of the foreign key of the **sub_accounts** table, the **ref_num** column, includes the ON DELETE CASCADE option. This option specifies that a delete of any row in the parent table **accounts** will automatically cause the corresponding rows of the child table **sub_accounts** to be deleted.

To delete a row from the **accounts** table that will cascade a delete to the **sub_accounts** table, you must turn on logging. After logging is turned on, you can delete the account number 2 from both tables, as the following example shows:

```
DELETE FROM accounts WHERE acc_num = 2
```

Restrictions on Cascading Deletes

You can use cascading deletes for most deletes, including deletes on self-referencing and cyclic queries. The only exception is correlated subqueries. In correlated subqueries, the subquery (or inner SELECT) is correlated when the value it produces depends on a value produced by the outer SELECT statement that contains it. If you have implemented cascading deletes, you cannot write deletes that use a child table in the correlated subquery. You receive an error when you attempt to delete from a correlated subquery.

Object Modes and Violation Detection

AD/XP

Dynamic Server with AD and XP Options does not support the violations table or the enabled, disabled, and filtering object modes. ♦

The object modes and violation detection features of the database can help you monitor data integrity. These features are particularly powerful when they are combined during schema changes or when insert, delete, and update operations are performed on large volumes of data over short periods.

You can use the object modes feature to change the modes of database objects. Database objects, within the context of a discussion of the object modes feature, are constraints, indexes, and triggers. Do not confuse database objects that are relevant to the object modes feature with generic database objects. Generic database objects are things like tables and synonyms. The database objects that relate specifically to object modes are constraints, indexes, and triggers and all of them have different modes.

Object Modes for Constraints

Constraints can have enabled, disabled, or filtering modes. The database manager does not enforce disabled constraints even though their definitions are still in the system catalog tables. Only constraints in the enabled and filtering mode are enforced. However, when a constraint is in filter mode, the database manager ensures the integrity of the base table for that particular constraint. The difference between enabled mode and filtering mode is apparent in the way the database manager handles a query that poses a violation of the constraint. The database manager uses the violation-detection feature when it deals with a constraint violation.

Consider an insert statement that violates a constraint. Depending on the mode of the constraint, the database manager handles the insert statement as follows:

- The constraint is enabled.

  An insert operation that violates an enabled constraint is not inserted into the target table. A constraint violation error is returned to the user, and effects of the statement are rolled back.
- The constraint is disabled.

  An insert operation that violates a disabled constraint is inserted in the target table, and no error is returned to the user.
- The constraint is filtering.

  An insert operation that violates a filtering constraint is not inserted into the target table; instead it is inserted into the violations table. The information about the integrity violation is created and stored in a third table called the diagnostics table. The effects of the insert operation are not rolled back. When you switch the mode of the constraint to filtering, you can determine whether or not an error is returned after a constraint is violated.

You can identify the reason for the failure when you analyze the information in the violations and diagnostic tables. You can then take corrective action or roll back the operation.

Object Modes for Unique Indexes

A unique index also has enabled, disabled, and filtering modes. A unique index in filtering mode operates the same way as a constraint in filtering mode. An index that does not avoid duplicate entries, however, only has enabled and disabled modes. When an index is disabled, its contents are not updated following insert, delete, or update modifications to the base table of the index. The optimizer cannot use a disabled index during a query because the index contents are not current.

Object Modes for Triggers

Unlike constraints and unique indexes, triggers have two modes. Formerly, a trigger either existed and was fired at the appropriate time by the database manager, or nothing happened because the trigger did not exist. Now you can use object modes to disable an existing trigger. The database manager ignores a trigger in disabled mode even though the catalog information of the disabled trigger is kept up to date. The database manager does not ignore a trigger in enabled mode. Triggers do not have a filtering mode since they do not impose any kind of integrity specification on the database.

AD/XP

Dynamic Server with AD and XP Options does not support SQL triggers. ♦

SQL Statements and Examples

For more information and examples, see the SET, START VIOLATIONS TABLE, and STOP VIOLATIONS TABLE statements in the *Informix Guide to SQL: Syntax*.

Interrupted Modifications

Even if all the software is error-free, and all the hardware is utterly reliable, the world outside the computer can interfere. Lightning might strike the building, interrupting the electrical supply and stopping the computer in the middle of your UPDATE statement. A more likely scenario occurs when a disk fills up, or a user supplies incorrect data, causing your multirow insert to stop early with an error. In any case, as you are modifying data, you must assume that some unforeseen event can interrupt the modification.

When a modification is interrupted by an external cause, you cannot be sure how much of the operation was completed. Even in a single-row operation, you cannot know whether the data reached the disk or the indexes were properly updated.

If multirow modifications are a problem, multistatement modifications are worse. They are usually embedded in programs so you do not see the individual SQL statements being executed. For example, to enter a new order in the demonstration database, you perform the following steps:

- Insert a row in the **orders** table. (This insert generates an order number.)
- For each item ordered, insert a row in the **items** table.

Two ways to program an order-entry application exist. One way is to make it completely interactive so that the program inserts the first row immediately, and then inserts each item as the user enters data. But this approach exposes the operation to the possibility of many more unforeseen events: the customer's telephone disconnecting, the user pressing the wrong key, the user's terminal or computer losing power, and so on.

The correct way to build an order-entry application is described in the following list:

- Accept all the data interactively.
- Validate the data, and expand it (look up codes in **stock** and **manufact**, for example).
- Display the information on the screen for inspection.
- Wait for the operator to make a final commitment.
- Perform the insertions quickly.

Even with these steps, an unforeseen circumstance can halt the program after it inserts the order but before it finishes inserting the items. If that happens, the database is in an unpredictable condition: its *data integrity* is compromised.

Transactions

The solution to all these potential problems is called the *transaction*. A transaction is a sequence of modifications that must be accomplished either completely or not at all. The database server guarantees that operations performed within the bounds of a transaction are either completely and perfectly committed to disk, or the database is restored to the same state as before the transaction started.

The transaction is not merely protection against unforeseen failures; it also offers a program a way to escape when the program detects a logical error.

Transaction Logging

The database server can keep a record of each change that it makes to the database during a transaction. If something happens to cancel the transaction, the database server automatically uses the records to reverse the changes. Many things can make a transaction fail. For example, the program that issues the SQL statements can crash or be terminated. As soon as the database server discovers that the transaction failed, which might be only after the computer and the database server are restarted, it uses the records from the transaction to return the database to the same state as before.

The process of keeping records of transactions is called *transaction logging* or simply *logging*. The records of the transactions, called *log records*, are stored in a portion of disk space separate from the database. This space is called the *logical log* because the log records represent logical units of the transactions.

AD/XP

Only Dynamic Server with AD and XP Options databases generate transaction records automatically. ♦

Most Informix databases do not generate transaction records automatically. The database administrator decides whether to make a database use transaction logging. Without transaction logging, you cannot roll back transactions.

AD/XP

Transaction Logging for Informix Dynamic Server with AD/XP Options

In addition to logical-log files, Dynamic Server with AD and XP Options allows you to create *logslices*. A logslice is a set of log files that occupy a dbslice. These log files are owned by multiple coservers, one log file per dbspace. Logslices simplify the process of adding and deleting log files because a logslice treats a set of log files as a single entity. For more information about logslices, see your *Administrator's Guide*.

Dynamic Server with AD and XP Options databases must be logged databases and logging cannot be turned off. However, you can specify that individual tables are logging or nonlogging tables. To meet the need for both logging and nonlogging tables, Dynamic Server with AD and XP Options supports the following types of permanent tables and temporary tables:

- Raw permanent tables (nonlogging)
- Static permanent tables (nonlogging)
- Operational permanent tables (logging)
- Standard permanent tables (logging)
- Scratch temporary tables (nonlogging)
- Temp temporary tables (logging)

For more information about the table types that Dynamic Server with AD and XP Options supports, see the *Informix Guide to Database Design and Implementation*.

Logging and Cascading Deletes

Logging must be turned on in your database for cascading deletes to work because, when you specify a cascading delete, the delete is first performed on the primary key of the parent table. If the system crashes after the rows of the primary key of the parent table are performed but before the rows of the foreign key of the child table are deleted, referential integrity is violated. If logging is turned off, even temporarily, deletes do not cascade. After logging is turned back on, however, deletes can cascade again.

IDS

With Dynamic Server, you turn logging on with the WITH LOG clause in the CREATE DATABASE statement. ♦

AD/XP

Databases that you create with Dynamic Server with AD and XP Options are always logging databases whether or not you include a WITH LOG clause in the CREATE DATABASE statement ♦

Specifying Transactions

You can use two methods to specify the boundaries of transactions with SQL statements. In the most common method, you specify the start of a multi-statement transaction by executing the BEGIN WORK statement. In databases that are created with the MODE ANSI option, no need exists to mark the beginning of a transaction. One is always in effect; you indicate only the end of each transaction.

In both methods, to specify the end of a successful transaction, execute the COMMIT WORK statement. This statement tells the database server that you reached the end of a series of statements that must succeed together. The database server does whatever is necessary to make sure that all modifications are properly completed and committed to disk.

A program can also cancel a transaction deliberately by executing the ROLLBACK WORK statement. This statement asks the database server to cancel the current transaction and undo any changes.

An order-entry application can use a transaction in the following ways when it creates a new order:

- Accept all data interactively
- Validate and expand it
- Wait for the operator to make a final commitment
- Execute BEGIN WORK
- Insert rows in the **orders** and **items** tables, checking the error code that the database server returns
- If no errors occurred, execute COMMIT WORK; otherwise execute ROLLBACK WORK

If any external failure prevents the transaction from being completed, the partial transaction rolls back when the system restarts. In all cases, the database is in a predictable state. Either the new order is completely entered, or it is not entered at all.

Backups and Logs with Informix Database Servers

By using transactions, you can ensure that the database is always in a consistent state and that your modifications are properly recorded on disk. But the disk itself is not perfectly safe. It is vulnerable to mechanical failures and to flood, fire, and earthquake. The only safeguard is to keep multiple copies of the data. These redundant copies are called *backup* copies.

The transaction log (also called the logical log) complements the backup copy of a database. Its contents are a history of all modifications that occurred since the last time the database was backed up. If you ever need to restore the database from the backup copy, you can use the transaction log to roll the database forward to its most recent state.

The database server contains elaborate features to support backups and logging. Your database server archive and backup guide describes these features.

The database server has very stringent requirements for performance and reliability (for example, it supports making backup copies while databases are in use).

The database server manages its own disk space, which is devoted to logging.

The database server performs logging concurrently for all databases using a limited set of log files. The log files can be copied to another medium (backed up) while transactions are active.

Database users never have to be concerned with these facilities because the database-server administrator usually manages them from a central location.

IDS

With Dynamic Server, you can use the **onunload** utility to make a personal backup copy of a single database or table. This program copies a table or a database to tape. Its output consists of binary images of the disk pages as they were stored in the database server. As a result, the copy can be made very quickly, and the corresponding **onload** program can restore the file quickly. However, the data format is not meaningful to any other programs. ♦

AD/XP

Dynamic Server with AD and XP Options does not support the **onload** and **onunload** utilities. To load or unload data, the database server uses *external tables*. For information about how to use external tables to load data, see your *Administrator's Guide*. ♦

If your database-server administrator uses ON-Bar to create backups and back up logical logs, you might also be able to create your own backup copies using ON-Bar. For more information, see your *Backup and Restore Guide*.

Concurrency and Locks

If your database is contained in a single-user workstation, without a network connecting it to other computers, concurrency is unimportant. In all other cases, you must allow for the possibility that, while your program is modifying data, another program is also reading or modifying the same data. *Concurrency* involves two or more independent uses of the same data at the same time.

A high level of concurrency is crucial to good performance in a multiuser database system. Unless controls exist on the use of data, however, concurrency can lead to a variety of negative effects. Programs could read obsolete data; modifications could be lost even though it seems they were entered successfully.

To prevent errors of this kind, the database server imposes a system of *locks*. A lock is a claim, or reservation, that a program can place on a piece of data. The database server guarantees that, as long as the data is locked, no other program can modify it. When another program requests the data, the database server either makes the program wait or turns it back with an error.

You use a combination of SQL statements to control the effect that locks have on your data access: SET LOCK MODE and either SET ISOLATION or SET TRANSACTION. You can understand the details of these statements after reading a discussion on the use of *cursors* from within programs. Cursors are covered in Chapter 5, “Programming with SQL,” and Chapter 6, “Modifying Data Through SQL Programs.” Also see Chapter 7, “Programming for a Multiuser Environment,” for more information about locking and concurrency.

Data Replication

Data replication, in the broadest sense of the term, means that database objects have more than one representation at more than one distinct site. For example, one way to replicate data, so that reports can be run against the data without disturbing client applications that are using the original database, is to copy the database to a database server on a different computer.

The following list describes the advantages of data replication:

- Clients who access replicated data locally, as opposed to remote data that is not replicated, experience improved performance because they do not have to use network services.
- Clients at all sites experience improved availability with replicated data, because if local replicated data is unavailable, a copy of the data is still available, albeit remotely.

AD/XP

Dynamic Server with AD and XP Options does not support data replication. ♦

These advantages do not come without a cost. Data replication obviously requires more storage for replicated data than for unreplicated data, and updating replicated data can take more processing time than updating a single object.

Data replication can actually be implemented in the logic of client applications, by explicitly specifying where data should be found or updated. However, this method of achieving data replication is costly, error-prone, and difficult to maintain. Instead, the concept of data replication is often coupled with *replication transparency*. Replication transparency is functionality built into a database server (instead of client applications) to handle the details of locating and maintaining data replicas automatically.

Informix Database Server Data Replication

Within the broad framework of data replication, an Informix database server implements nearly transparent data replication of entire database servers. All the data managed by one Informix database server is replicated and dynamically updated on another Informix database server, usually at a remote site. Data replication of an Informix database server is sometimes called *hot site backup*, because it provides a means of maintaining a backup copy of the entire database server that can be used quickly in the event of a catastrophic failure.

Because the database server provides replication transparency, you generally do not need to be concerned with or aware of data replication; the database server administrator takes care of it. However, if your organization decides to use data replication, you should be aware that special connectivity considerations exist for client applications in a data replication environment. These considerations are described in your *Administrator's Guide*.

The Informix Enterprise Replication feature provides a different method of data replication. For information on this feature, see the *Guide to Informix Enterprise Replication*.

AD/XP

Dynamic Server with AD and XP Options does not support the Enterprise Replication feature. ♦

Summary

Database access is regulated by the privileges that the database owner grants to you. The privileges that let you query data are often granted automatically, but the ability to modify data is regulated by specific Insert, Delete, and Update privileges that are granted on a table-by-table basis.

If data integrity constraints are imposed on the database, your ability to modify data is restricted by those constraints. Your database- and table-level privileges, along with any data constraints, control how and when you can modify data. In addition, the object modes and violation detection features of the database affect how you can modify data and help to preserve the integrity of your data.

You can delete one or more rows from a table with the DELETE statement. Its WHERE clause selects the rows; use a SELECT statement with the same clause to preview the deletes.

Rows are added to a table with the INSERT statement. You can insert a single row that contains specified column values, or you can insert a block of rows that a SELECT statement generates.

You use the UPDATE statement to modify the contents of existing rows. You specify the new contents with expressions that can include subqueries, so that you can use data that is based on other tables or the updated table itself. The statement has two forms. In the first form, you specify new values column by column. In the second form, a SELECT statement or a record variable generates a set of new values.

You use the REFERENCES clause of the CREATE TABLE and ALTER TABLE statements to create relationships between tables. The ON DELETE CASCADE option of the REFERENCES clause allows you to delete rows from parent and associated child tables with one DELETE statement.

You use transactions to prevent unforeseen interruptions in a modification from leaving the database in an indeterminate state. When modifications are performed within a transaction, they are rolled back after an error occurs. The transaction log also extends the periodically made backup copy of the database. If the database must be restored, it can be brought back to its most recent state.

Data replication, which is transparent to users, offers another type of protection from catastrophic failures.

Chapter

5 Programming with SQL

SQL in Programs 5-4
    SQL in SQL APIs 5-4
    SQL in Application Languages 5-5
    Static Embedding 5-5
    Dynamic Statements 5-5
    Program Variables and Host Variables 5-6

Calling the Database Server 5-8
    SQL Communications Area 5-8
    SQLCODE Field 5-9
        End of Data. 5-9
        Negative Codes 5-10
    SQLERRD Array 5-10
    SQLWARN Array 5-11
    SQLERRM Character Array 5-13
    SQLSTATE Value 5-13

Retrieving Single Rows 5-14
    Data-Type Conversion 5-15
    Working with Null Data. 5-16
    Dealing with Errors 5-17
        End of Data. 5-17
        End of Data with Databases That Are Not ANSI Compliant . . 5-18
        Serious Errors 5-18
        Interpreting End of Data with Aggregate Functions 5-18
        Using Default Values 5-19

Retrieving Multiple Rows 5-20
    Declaring a Cursor 5-20
    Opening a Cursor 5-21
    Fetching Rows 5-22
        Detecting End of Data 5-22

Locating the INTO Clause 5-23
Cursor Input Modes 5-23
Active Set of a Cursor. 5-24
Creating the Active Set 5-24
Active Set for a Sequential Cursor 5-25
Active Set for a Scroll Cursor 5-25
Active Set and Concurrency 5-26
Using a Cursor: A Parts Explosion 5-27

Dynamic SQL 5-29
Preparing a Statement 5-30
Executing Prepared SQL. 5-31
Dynamic Host Variables 5-32
Freeing Prepared Statements 5-33
Quick Execution 5-33

Embedding Data-Definition Statements 5-34

Embedding Grant and Revoke Privileges 5-34

Summary 5-37

In the examples in the previous chapters, SQL is treated as if it were an interactive computer language; that is, as if you could type a SELECT statement directly into the database server and see rows of data rolling back to you.

Of course, that is not the case. Many layers of software stand between you and the database server. The database server retains data in a binary form that must be formatted before it can be displayed. It does not return a mass of data at once; it returns one row at a time, as a program requests it.

You can access information in your database in several ways:

- Through interactive access with DB-Access or Relational Object Manager
- Through application programs written with an SQL API such as ESQL/C
- Through an application language such as Stored Procedure Language (SPL)

Almost any program can contain SQL statements, execute them, and retrieve data from a database server. This chapter explains how these activities are performed and indicates how you can write programs that perform them.

This chapter is only an introduction to the concepts that are common to SQL programming in any language. Before you can write a successful program in a particular programming language, you must first become fluent in that language. Then, because the details of the process are different in every language, you must become familiar with the manual for the Informix SQL API specific to that language.

SQL in Programs

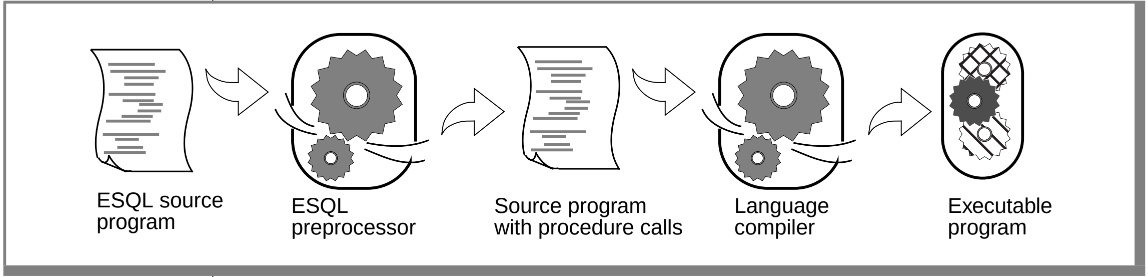
You can write a program in any of several languages and mix SQL statements among the other statements of the program, just as if they were ordinary statements of that programming language. These SQL statements are *embedded* in the program, and the program contains *embedded SQL*, which Informix often abbreviates as ESQL.

SQL in SQL APIs

ESQL products are Informix SQL APIs (application programming interfaces). Informix produces an SQL API for the C programming language.

Figure 5-1 shows how an SQL API product works. You write a source program in which you treat SQL statements as executable code. Your source program is processed by an embedded SQL *preprocessor*, a program that locates the embedded SQL statements and converts them into a series of procedure calls and special data structures.

Figure 5-1
Overview of Processing a Program with Embedded SQL Statements



The converted source program then passes through the programming language compiler. The compiler output becomes an executable program after it is linked with a static or *dynamic* library of SQL API procedures. When the program runs, the SQL API library procedures are called; they set up communication with the database server to carry out the SQL operations.

If you link your executable program to a threading library package, you can develop ESQL/C *multithreaded applications*. A multithreaded application can have many threads of control. It separates a process into multiple execution threads, each of which runs independently. The major advantage of a multithreaded ESQL/C application is that each thread can have many active connections to a database server simultaneously. While a nonthreaded ESQL/C application can establish many connections to one or more databases, it can have only one connection active at a time. A multithreaded ESQL/C application can have one active connection per thread and many threads per application.

For more information on multithreaded applications, see the *INFORMIX-ESQL/C Programmer's Manual*.

SQL in Application Languages

Whereas an SQL API product allows you to embed SQL in the host language, some languages include SQL as a natural part of their statement set. Informix Stored Procedure Language (SPL) uses SQL as a natural part of its statement set. You use an SQL API product to write application programs. You use SPL to write procedures that are stored with a database and called from an application program.

Static Embedding

You can introduce SQL statements into a program through static embedding or dynamic statements. The simpler and more common way is by *static embedding*, which means that the SQL statements are written as part of the code. The statements are *static* because they are a fixed part of the source text. For further information on static embedding, see "Retrieving Single Rows" on page 5-14 and "Retrieving Multiple Rows" on page 5-20.

Dynamic Statements

Some applications require the ability to compose SQL statements *dynamically*, in response to user input. For example, a program might have to select different columns or apply different criteria to rows, depending on what the user wants.

With *dynamic* SQL, the program composes an SQL statement as a string of characters in memory and passes it to the database server to be executed. Dynamic statements are not part of the code; they are constructed in memory during execution.

For further information on dynamic SQL statements, see "Dynamic SQL" on page 5-29.

Program Variables and Host Variables

Application programs can use program variables within SQL statements. In SPL, you put the program variable in the SQL statement as syntax allows. For example, a DELETE statement can use a program variable in its WHERE clause.

The following code example shows a program variable in SPL:

```
CREATE PROCEDURE delete_item (drop_number INT)
.
.
.
DELETE FROM items WHERE order_num = drop_number
.
.
.
```

In applications that use embedded SQL statements, the SQL statements can refer to the contents of program variables. A program variable that is named in an embedded SQL statement is called a *host variable* because the SQL statement is thought of as a *guest* in the program.

The following example shows a DELETE statement as it might appear when it is embedded in an INFORMIX-ESQL/C source program:

```
EXEC SQL delete FROM items
    WHERE order_num = :onum;
```

In this program, you see an ordinary DELETE statement, as described in Chapter 4, "Modifying Data." When the ESQL/C program is executed, a row of the **items** table is deleted; multiple rows can also be deleted.

The statement contains one new feature. It compares the **order_num** column to an item written as **:onum**, which is the name of a host variable.

An SQL API product provides a way to delimit the names of host variables when they appear in the context of an SQL statement. In ESQL/C, a host variable can be introduced with either a dollar sign ($) or a colon (:). The colon is the ANSI-compatible format. The example statement asks the database server to delete rows in which the order number equals the current contents of the host variable named **:onum**. This numeric variable was declared and assigned a value earlier in the program.

In INFORMIX-ESQL/C, an SQL statement can be introduced with either a leading dollar sign ($) or the words EXEC SQL.

The differences of syntax as illustrated in the preceding examples are trivial; the essential point is that the SQL API and SPL languages let you perform the following tasks:

- Embed SQL statements in a source program as if they were executable statements of the host language.
- Use program variables in SQL expressions the way literal values are used.

If you have programming experience, you can immediately see the possibilities. In the example, the order number to be deleted is passed in the variable **onum**. That value comes from any source that a program can use. It can be read from a file, the program can prompt a user to enter it, or it can be read from the database. The DELETE statement itself can be part of a subroutine (in which case **onum** can be a parameter of the subroutine); the subroutine can be called once or repetitively.

In short, when you embed SQL statements in a program, you can apply all the power of the host language to them. You can hide the SQL statements under many interfaces, and you can embellish the SQL functions in many ways.

Calling the Database Server

Executing an SQL statement is essentially calling the database server as a subroutine. Information must pass from the program to the database server, and information must be returned from the database server to the program.

Some of this communication is done through host variables. You can think of the host variables named in an SQL statement as the parameters of the procedure call to the database server. In the preceding example, a host variable acts as a parameter of the WHERE clause. Host variables receive data that the database server returns, as described in "Retrieving Multiple Rows" on page 5-20.

SQL Communications Area

The database server always returns a result code, and possibly other information about the effect of an operation, in a data structure known as the SQL Communications Area (SQLCA). If the database server executes an SQL statement in a stored procedure, the SQLCA of the calling application contains the values triggered by the SQL statement in the procedure.

The principal fields of the SQLCA are listed in Figure 5-2 through Figure 5-4. The syntax that you use to describe a data structure such as the SQLCA, as well as the syntax that you use to refer to a field in it, differs among programming languages. For details, see your SQL API manual.

In particular, the subscript by which you name one element of the SQLERRD and SQLWARN arrays differs. Array elements are numbered starting with zero in INFORMIX-ESQL/C, but starting with one in other languages. In this discussion, the fields are named with specific words such as *third*, and you must translate these words into the syntax of your programming language.

You can also use the SQLSTATE variable of the GET DIAGNOSTICS statement to detect, handle, and diagnose errors. See "SQLSTATE Value" on page 5-13.

SQLCODE Field

The SQLCODE field is the primary return code of the database server. After every SQL statement, SQLCODE is set to an integer value as Figure 5-2 shows. When that value is zero, the statement is performed without error. In particular, when a statement is supposed to return data into a host variable, a code of zero means that the data has been returned and can be used. Any nonzero code means the opposite. No useful data was returned to host variables.

Figure 5-2
Values of SQLCODE

Return value	Interpretation
value < 0	Specifies an error code.
value = 0	Indicates success.
0 < *value* < 100	After a DESCRIBE statement, an integer value that represents the type of SQL statement that is described.
100	After a successful query that returns no rows, indicates the NOT FOUND condition. NOT FOUND can also occur in an ANSI-compliant database after an INSERT INTO / SELECT, UPDATE, DELETE, or SELECT... INTO TEMP statement fails to access any rows.

End of Data

The database server sets SQLCODE to `100` when the statement is performed correctly but no rows are found. This condition can occur in two situations.

The first situation involves a query that uses a cursor. (Queries that use cursors are described under "Retrieving Multiple Rows" on page 5-20.) In these queries, the FETCH statement retrieves each value from the active set into memory. After the last row is retrieved, a subsequent FETCH statement cannot return any data. When this condition occurs, the database server sets SQLCODE to `100`, which indicates *end of data, no rows found*.

The second situation involves a query that does not use a cursor. In this case, the database server sets SQLCODE to `100` when no rows satisfy the query condition. In databases that are not ANSI compliant, only a SELECT statement that returns no rows causes SQLCODE to be set to `100`.

ANSI

In ANSI-compliant databases, SELECT, DELETE, UPDATE, and INSERT statements all set SQLCODE to `100` if no rows are returned. ♦

Negative Codes

When something unexpected goes wrong during a statement, the database server returns a negative number in SQLCODE to explain the problem. The meanings of these codes are documented in the *Informix Error Messages* manual and in the on-line error message file.

SQLERRD Array

Some error codes that can be reported in SQLCODE reflect general problems. The database server can set a more detailed code in the second field of SQLERRD that reveals the error encountered by the database server I/O routines or by the operating system.

The integers in the SQLERRD array are set to different values following different statements. The first and fourth elements of the array are used only in INFORMIX-ESQL/C. Figure 5-3 shows how the fields are used.

These additional details can be very useful. For example, you can use the value in the third field to report how many rows were deleted or updated. When your program prepares an SQL statement that is entered by the user, and an error is found, the value in the fifth field enables you to display the exact point of error to the user. (DB-Access and Relational Object Manager use this feature to position the cursor when you ask to modify a statement after an error.)

Figure 5-3
Fields of SQLERRD

Field	Interpretation
First	After a successful PREPARE statement for a SELECT, UPDATE, INSERT, or DELETE statement, or after a select cursor is opened, this field contains the estimated number of rows affected
Second	When **SQLCODE** contains an error code, this field contains either zero or an additional error code, called the ISAM error code, that explains the cause of the main error. After a successful insert operation of a single row, this field contains the value of any SERIAL value generated for that row
Third	After a successful multirow insert, update, or delete operation, this field contains the number of rows that were processed. After a multirow insert, update, or delete operation that ends with an error, this field contains the number of rows that were successfully processed before the error was detected.
Fourth	After a successful PREPARE statement for a SELECT, UPDATE, INSERT, or DELETE statement, or after a select cursor has been opened, this field contains the estimated weighted sum of disk accesses and total rows processed.
Fifth	After a syntax error in a PREPARE, EXECUTE IMMEDIATE, DECLARE, or static SQL statement, this field contains the offset in the statement text where the error was detected.
Sixth	After a successful fetch of a selected row, or a successful insert, update, or delete operation, this field contains the rowid (physical address) of the last row that was processed. Whether this rowid value corresponds to a row that the database server returns to the user depends on how the database server processes a query, particularly for SELECT statements.

SQLWARN Array

The eight character fields in the SQLWARN array are set to either a blank or to `W` to indicate a variety of special conditions. Their meanings depend on the statement just executed.

A set of warning flags appears when a database opens, that is, following a CONNECT, DATABASE or CREATE DATABASE statement. These flags tell you some characteristics of the database as a whole.

A second set of flags appears following any other statement. These flags reflect unusual events that occur during the statement, which are usually not serious enough to be reflected by SQLCODE.

Both sets of SQLWARN values are summarized in Figure 5-4.

Figure 5-4
Fields of SQLWARN

Field	When Opening or Connecting to a Database:	All Other Operations:
First	Set to W when any other warning field is set to W. If blank, others need not be checked.	Set to W when any other warning field is set to W.
Second	Set to W when the database now open uses a transaction log.	Set to W if a column value is truncated when it is fetched into a host variable using a FETCH or a SELECT...INTO statement. On a REVOKE ALL statement, set to W when not all seven table-level privileges are revoked.
Third	Set to W when the database now open is ANSI compliant.	Set to W when a FETCH or SELECT statement returns an aggregate function (SUM, AVG, MIN, MAX) value that is null.
Fourth	Set to W when the database server is Informix Dynamic Server with Universal Data Option.	On a SELECT...INTO, FETCH...INTO, or EXECUTE...INTO statement, set to W when the number of items in the select list is not the same as the number of host variables given in the INTO clause to receive them. On a GRANT ALL statement, set to W when not all seven table-level privileges are granted.
Fifth	Set to W when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).	Set to W after a DESCRIBE statement if the prepared statement contains a DELETE statement or an UPDATE statement without a WHERE clause.

(1 of 2)

Field	When Opening or Connecting to a Database:	All Other Operations:
Sixth	Set to W when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).	Set to W following execution of a statement that does not use ANSI-standard SQL syntax (provided the **DBANSIWARN** environment variable is set).
Seventh	Set to W when the application is connected to a database server that is running in secondary mode. The database server is a secondary server in a data-replication pair (that is, the server is available only for read operations).	Set to W when a data fragment (a dbspace) has been skipped during query processing (when the DATASKIP feature is on).
Eighth	Set to W when client **DB_LOCALE** does not match the database locale. For more information, see the *Informix Guide to GLS Functionality*.	Reserved.

(2 of 2)

SQLERRM Character Array

The SQLERRM array is a 71-character array that contains the variable, such as a table name, that is placed in the error message. For some networked applications, it contains an error message generated by networking software.

SQLSTATE Value

Certain Informix products, such as INFORMIX-ESQL/C, support the SQLSTATE value in compliance with X/Open and ANSI SQL standards. The GET DIAGNOSTICS statement reads the SQLSTATE value to diagnose errors after you run an SQL statement. The database server returns a result code in a five-character string that is stored in a variable called SQLSTATE. The SQLSTATE error code, or value, tells you the following information about the most recently executed SQL statement:

- If the statement was successful
- If the statement was successful but generated warnings
- If the statement was successful but generated no data
- If the statement failed

For more information on the GET DIAGNOSTICS statement, the SQLSTATE variable, and the meaning of the SQLSTATE return codes, see the GET DIAGNOSTICS statement in the *Informix Guide to SQL: Syntax*.

Tip: *If your Informix product supports GET DIAGNOSTICS and SQLSTATE, Informix recommends that you use them as the primary structure to detect, handle, and diagnose errors. Using SQLSTATE allows you to detect multiple errors, and it is ANSI compliant.*

Retrieving Single Rows

You can use embedded SELECT statements to retrieve single rows from the database into host variables. When a SELECT statement returns more than one row of data, however, a program must use a more complicated method to fetch the rows one at a time. Multiple-row select operations are discussed in "Retrieving Multiple Rows" on page 5-20.

To retrieve a single row of data, simply embed a SELECT statement in your program. The following example shows how you can write the embedded SELECT statement using INFORMIX-ESQL/C:

```
EXEC SQL select avg (total_price)
    into :avg_price
    from items
    where order_num in
        (select order_num from orders
        where order_date < date('6/1/94'));
```

The INTO clause is the only detail that distinguishes this statement from any example in Chapter 2, "Composing Simple SELECT Statements," or Chapter 3, "Composing Advanced SELECT Statements." This clause specifies the host variables that are to receive the data that is produced.

When the program executes an embedded SELECT statement, the database server performs the query. The example statement selects an aggregate value, so that it produces exactly one row of data. The row has only a single column, and its value is deposited in the host variable named **avg_price**. Subsequent lines of the program can use that variable.

You can use statements of this kind to retrieve single rows of data into host variables. The single row can have as many columns as desired. If a query produces more than one row of data, the database server cannot return any data. It returns an error code instead.

You should list as many host variables in the INTO clause as there are items in the select list. If, by accident, these lists are of different lengths, the database server returns as many values as it can and sets the warning flag in the fourth field of SQLWARN.

Data-Type Conversion

The following ESQL/C example retrieves the average of a DECIMAL column, which is itself a DECIMAL value. However, the host variable into which the average of the DECIMAL column is placed is *not* required to have that data type.

```
EXEC SQL select avg (total_price) into :avg_price
    from items;
```

The declaration of the receiving variable **avg_price** in the previous example of ESQL/C code is not shown. The declaration could be any one of the following definitions:

```
int avg_price;
double avg_price;
char avg_price[16];
dec_t avg_price; /* typedef of decimal number structure */
```

The data type of each host variable that is used in a statement is noted and passed to the database server along with the statement. The database server does its best to convert column data into the form that the receiving variables use. Almost any conversion is allowed, although some conversions cause a precision loss. The results of the preceding example differ, depending on the data type of the receiving host variable, as the following table shows.

Data Type	Result
FLOAT	The database server converts the decimal result to FLOAT, possibly truncating some fractional digits. If the magnitude of a decimal exceeds the maximum magnitude of the FLOAT format, an error is returned.
INTEGER	The database server converts the result to INTEGER, truncating fractional digits if necessary. If the integer part of the converted number does not fit the receiving variable, an error occurs.
CHARACTER	The database server converts the decimal value to a CHARACTER string. If the string is too long for the receiving variable, it is truncated. The second field of SQLWARN is set to `W` and the value in the SQLSTATE variable is `01004`.

Working with Null Data

What if the program retrieves a null value? Null values can be stored in the database, but the data types supported by programming languages do not recognize a null state. A program must have some way to recognize a null item to avoid processing it as data.

Indicator variables meet this need in SQL APIs. An indicator variable is an additional variable that is associated with a host variable that might receive a null item. When the database server puts data in the main variable, it also puts a special value in the indicator variable to show whether the data is null. In the following INFORMIX-ESQL/C example, a single row is selected, and a single value is retrieved into the host variable **op_date**:

```
EXEC SQL select paid_date
        into :op_date:op_d_ind
        from orders
        where order_num = $the_order;
if (op_d_ind < 0) /* data was null */
    rstrdate ('01/01/1900', :op_date);
```

Because the value might be null, an indicator variable named **op_d_ind** is associated with the host variable. (It must be declared as a short integer elsewhere in the program.)

Following execution of the SELECT statement, the program tests the indicator variable for a negative value. A negative number (usually `-1`) means that the value retrieved into the main variable is null. If the variable is null, this program uses an ESQL/C library function to assign a default value to the host variable. (The function **rstrdate** is part of the INFORMIX-ESQL/C product.)

The syntax that you use to associate an indicator variable with a host variable differs with the language you are using, but the principle is the same in all languages.

Dealing with Errors

Although the database server handles conversion between data types automatically, several things still can go wrong with a SELECT statement. In SQL programming, as in any kind of programming, you must anticipate errors and provide for them at every point.

End of Data

One common event is that no rows satisfy a query. This event is signalled by an SQLSTATE code of `02000` and by a code of `100` in SQLCODE after a SELECT statement. This code indicates an error or a normal event, depending entirely on your application. If you are sure a row or rows should satisfy the query (for example, if you are reading a row using a key value that you just read from a row of another table), then the end-of-data code represents a serious failure in the logic of the program. On the other hand, if you select a row based on a key that is supplied by a user or by some other source that is less reliable than a program, a lack of data can be a normal event.

End of Data with Databases That Are Not ANSI Compliant

If your database is not ANSI compliant, the end-of-data return code, 100, is set in SQLCODE following SELECT statements only. In addition, the SQLSTATE value is set to 02000. (Other statements, such as INSERT, UPDATE, and DELETE, set the third element of SQLERRD to show how many rows they affected; this topic is covered in Chapter 6, “Modifying Data Through SQL Programs.”)

Serious Errors

Errors that set SQLCODE to a negative value or SQLSTATE to a value that begins with anything other than 00, 01, or 02 are usually serious. Programs that you have developed and that are in production should rarely report these errors. Nevertheless, it is difficult to anticipate every problematic situation, so your program must be able to deal with these errors.

For example, a query can return error -206, which means that a table specified in the query is not in the database. This condition occurs if someone dropped the table after the program was written, or if the program opened the wrong database through some error of logic or mistake in input.

Interpreting End of Data with Aggregate Functions

A SELECT statement that uses an aggregate function such as SUM, MIN, or AVG always succeeds in returning at least one row of data, even when no rows satisfy the WHERE clause. An aggregate value based on an empty set of rows is null, but it exists nonetheless.

However, an aggregate value is also null if it is based on one or more rows that all contain null values. If you must be able to detect the difference between an aggregate value that is based on no rows and one that is based on some rows that are all null, you must include a COUNT function in the statement and an indicator variable on the aggregate value. You can then work out the following cases.

Count Value	Indicator	Case
0	-1	Zero rows selected
>0	-1	Some rows selected; all were null
>0	0	Some non-null rows selected

Using Default Values

You can handle these inevitable errors in many ways. In some applications, more lines of code are used to handle errors than to execute functionality. In the examples in this section, however, one of the simplest solutions, the default value, should work, as the following example shows:

```
avg_price = 0; /* set default for errors */
EXEC SQL select avg (total_price)
        into :avg_price:null_flag
        from items;
if (null_flag < 0) /* probably no rows */
    avg_price = 0; /* set default for 0 rows */
```

The previous example deals with the following considerations:

- If the query selects some non-null rows, the correct value is returned and used. This result is the expected and most frequent one.
- If the query selects no rows, or in the much less likely event that it selects only rows that have null values in the **total_price** column (a column that should never be null), the indicator variable is set, and the default value is assigned.
- If any serious error occurs, the host variable is left unchanged; it contains the default value initially set. At this point in the program, the programmer sees no need to trap such errors and report them.

Retrieving Multiple Rows

When any chance exists that a query could return more than one row, the program must execute the query differently. Multirow queries are handled in two stages. First, the program starts the query. (No data is returned immediately.) Then the program requests the rows of data one at a time.

These operations are performed using a special data object called a *cursor*. A cursor is a data structure that represents the current state of a query. The following list shows the general sequence of program operations:

1. The program *declares* the cursor and its associated SELECT statement, which merely allocates storage to hold the cursor.
2. The program *opens* the cursor, which starts the execution of the associated SELECT statement and detects any errors in it.
3. The program *fetches* a row of data into host variables and processes it.
4. The program *closes* the cursor after the last row is fetched.
5. When the cursor is no longer needed, the program *frees* the cursor to deallocate the resources it uses.

These operations are performed with SQL statements named DECLARE, OPEN, FETCH, CLOSE, and FREE.

Declaring a Cursor

You use the DECLARE statement to declare a cursor. This statement gives the cursor a name, specifies its use, and associates it with a statement. The following example is written in INFORMIX-ESQL/C:

```
EXEC SQL DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
    INTO o_num, i_num, s_num
    FROM items
    FOR READ ONLY;
```

The declaration gives the cursor a name (**the_item** in this case) and associates it with a SELECT statement. (Chapter 6, "Modifying Data Through SQL Programs," discusses how a cursor can also be associated with an INSERT statement.)

The SELECT statement in this example contains an INTO clause. The INTO clause specifies which variables receive data. You can also use the FETCH statement to specify which variables receive data, as discussed in "Locating the INTO Clause" on page 5-23.

The DECLARE statement is not an active statement; it merely establishes the features of the cursor and allocates storage for it. You can use the cursor declared in the preceding example to read through the **items** table once. Cursors can be declared to read backward and forward (see "Cursor Input Modes" on page 5-23). This cursor, because it lacks a FOR UPDATE clause and because it is designated FOR READ ONLY, is used only to read data, not to modify it. (The use of cursors to modify data is covered in Chapter 6, "Modifying Data Through SQL Programs.")

Opening a Cursor

The program opens the cursor when it is ready to use it. The OPEN statement activates the cursor. It passes the associated SELECT statement to the database server, which begins the search for matching rows. The database server processes the query to the point of locating or constructing the first row of output. It does not actually return that row of data, but it does set a return code in SQLSTATE and in SQLCODE for SQL APIs. The following example shows the OPEN statement in ESQL/C:

```
EXEC SQL OPEN the_item;
```

Because the database server is seeing the query for the first time, it might detect a number of errors. After the program opens the cursor, it should test SQLSTATE or SQLCODE. If the SQLSTATE value is greater than `02000`, or the SQLCODE contains a negative number, the cursor is not usable. An error might be present in the SELECT statement, or some other problem might prevent the database server from executing the statement.

If SQLSTATE is equal to `00000`, or SQLCODE contains a zero, the SELECT statement is syntactically valid, and the cursor is ready to use. At this point, however, the program does not know if the cursor can produce any rows.

Fetching Rows

The program uses the FETCH statement to retrieve each row of output. This statement names a cursor and can also name the host variables that receive the data. The following example shows the completed INFORMIX-ESQL/C code:

```
EXEC SQL DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        INTO :o_num, :i_num, :s_num
        FROM items;
EXEC SQL OPEN the_item;
while(SQLCODE == 0)
{
    EXEC SQL FETCH the_item;
    if(SQLCODE == 0)
        printf("%d, %d, %d", o_num, i_num, s_num);
}
```

Detecting End of Data

In the previous example, the WHILE condition prevents execution of the loop in case the OPEN statement returns an error. The same condition terminates the loop when SQLCODE is set to 100 to signal the end of data. However, the loop contains a test of SQLCODE. This test is necessary because, if the SELECT statement is valid yet finds no matching rows, the OPEN statement returns a zero, but the first fetch returns 100 (end of data) and no data. The following example shows another way to write the same loop:

```
EXEC SQL DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
    INTO :o_num, :i_num, :s_num
    FROM items;
EXEC SQL OPEN the_item;
if(SQLCODE == 0)
    EXEC SQL FETCH the_item;        /* fetch 1st row
while(SQLCODE == 0)
{
    printf("%d, %d, %d", o_num, i_num, s_num);
    EXEC SQL FETCH the_item;
}
```

In this version, the case of no returned rows is handled early, so no second test of SQLCODE exists within the loop. These versions have no measurable difference in performance because the time cost of a test of SQLCODE is a tiny fraction of the cost of a fetch.

Locating the INTO Clause

The INTO clause names the host variables that are to receive the data returned by the database server. The INTO clause must appear in either the SELECT or the FETCH statement. However it cannot appear in both statements. The following example specifies host variables in the FETCH statement:

```
EXEC SQL DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        FROM items;
EXEC SQL OPEN the_item;
while(SQLCODE == 0)
{
    EXEC SQL FETCH the_item INTO :o_num, :i_num, :s_num;
    if(SQLCODE == 0)
        printf("%d, %d, %d", o_num, i_num, s_num);
}
```

This form lets you fetch different rows into different locations. For example, you could use this form to fetch successive rows into successive elements of an array.

Cursor Input Modes

For purposes of input, a cursor operates in one of two modes, *sequential* or *scrolling*. A sequential cursor can fetch only the next row in sequence, so a sequential cursor can read through a table only once each time the cursor is opened. A scroll cursor can fetch the next row or any of the output rows, so a scroll cursor can read the same rows multiple times. The following example shows a sequential cursor declared in INFORMIX-ESQL/C:

```
EXEC SQL declare pcurs cursor for
    select customer_num, lname, city
        from customer;
```

After the cursor is opened, it can be used only with a sequential fetch that retrieves the next row of data, as the following example shows.

```
EXEC SQL fetch p_curs into:cnum, :clname, :ccity;
```

Each sequential fetch returns a new row.

A scroll cursor is declared with the keywords SCROLL CURSOR, as the following example from INFORMIX-ESQL/C shows:

```
EXEC SQL DECLARE s_curs SCROLL CURSOR FOR
    SELECT order_num, order_date FROM orders
        WHERE customer_num > 104
```

Use the scroll cursor with a variety of fetch options. For example, the ABSOLUTE option specifies the absolute row position of the row to fetch.

```
EXEC SQL FETCH ABSOLUTE :numrow s_curs
    INTO :nordr, :nodat
```

This statement fetches the row whose position is given in the host variable **numrow**. You can also fetch the current row again, or you can fetch the first row and then scan through all the rows again. However, these features have a price, as the next section describes. For additional options that apply to scroll cursors, see the FETCH statement in the *Informix Guide to SQL: Syntax*.

Active Set of a Cursor

Once a cursor is opened, it stands for some selection of rows. The set of all rows that the query produces is called the *active set* of the cursor. It is easy to think of the active set as a well-defined collection of rows and to think of the cursor as pointing to one row of the collection. This situation is true as long as no other programs are modifying the same data concurrently.

Creating the Active Set

When a cursor is opened, the database server does whatever is necessary to locate the first row of selected data. Depending on how the query is phrased, this action can be very easy, or it can require a great deal of work and time. Consider the following declaration of a cursor:

```
EXEC SQL DECLARE easy CURSOR FOR
    SELECT fname, lname FROM customer
        WHERE state = 'NJ'
```

Because this cursor queries only a single table in a simple way, the database server quickly determines whether any rows satisfy the query and identifies the first one. The first row is the only row the cursor finds at this time. The rest of the rows in the active set remain unknown. As a contrast, consider the following declaration of a cursor:

```
EXEC SQL DECLARE hard SCROLL CURSOR FOR
    SELECT C.customer_num, O.order_num, sum (items.total_price)
        FROM customer C, orders O, items I
        WHERE C.customer_num = O.customer_num
            AND O.order_num = I.order_num
            AND O.paid_date is null
        GROUP BY C.customer_num, O.order_num
```

The active set of this cursor is generated by joining three tables and grouping the output rows. The optimizer might be able to use indexes to produce the rows in the correct order, but generally the use of ORDER BY or GROUP BY clauses requires the database server to generate all the rows, copy them to a temporary table, and sort the table, before it can determine which row to present first.

In cases where the active set is entirely generated and saved in a temporary table, the database server can take quite some time to open the cursor. Afterwards, the database server could tell the program exactly how many rows the active set contains. However, this information is not made available. One reason is that you can never be sure which method the optimizer uses. If the optimizer can avoid sorts and temporary tables, it does so; but very small changes in the query, in the sizes of the tables, or in the available indexes can change the optimizer's methods.

Active Set for a Sequential Cursor

The database server attempts to use as few resources as possible to maintain the active set of a cursor. If it can do so, the database server never retains more than the single row that is fetched next. It can do this for most sequential cursors. On each fetch, it returns the contents of the current row and locates the next one.

Active Set for a Scroll Cursor

All the rows in the active set for a scroll cursor must be retained until the cursor closes because the database server cannot be sure which row the program will ask for next.

Most frequently, the database server implements the active set of a scroll cursor as a temporary table. The database server might not fill this table immediately, however (unless it created a temporary table to process the query). Usually it creates the temporary table when the cursor is opened. Then, the first time a row is fetched, the database server copies it into the temporary table and returns it to the program. When a row is fetched for a second time, it can be taken from the temporary table. This scheme uses the fewest resources in the event that the program abandons the query before it fetches all the rows. Rows that are never fetched are not created or saved.

Active Set and Concurrency

When only one program is using a database, the members of the active set cannot change. This situation describes most personal computers, and it is the easiest situation to think about. But some programs must be designed for use in a multiprogramming system, where two, three, or dozens of different programs can work on the same tables simultaneously.

When other programs can update the tables while your cursor is open, the idea of the active set becomes less useful. Your program can see only one row of data at a time, but all other rows in the table can be changing.

In the case of a simple query, when the database server holds only one row of the active set, any other row can change. The instant after your program fetches a row, another program can delete the same row or update it so that if it is examined again, it is no longer part of the active set.

When the active set, or part of it, is saved in a temporary table, *stale data* can present a problem. That is, the rows in the actual tables, from which the active-set rows are derived, can change. If they do, some of the active-set rows no longer reflect the current table contents.

These ideas seem unsettling at first, but as long as your program only reads the data, stale data does not exist, or rather, all data is equally stale. The active set is a snapshot of the data as it is at one moment in time. A row is different the next day; it does not matter if it is also different in the next millisecond. To put it another way, no practical difference exists between changes that occur while the program is running and changes that are saved and applied the instant that the program terminates.

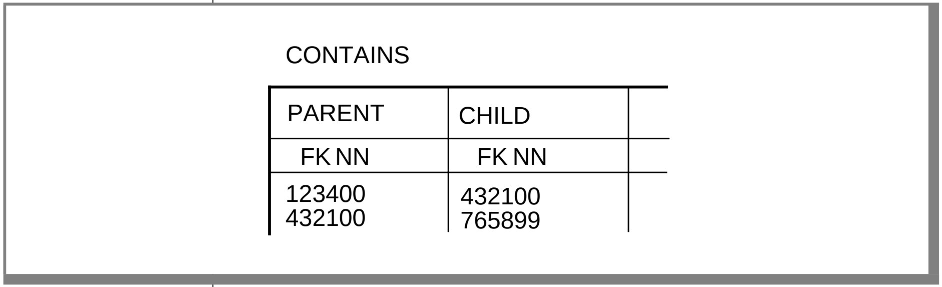
The only time that stale data can cause a problem is when the program intends to use the input data to modify the same database; for example, when a banking application must read an account balance, change it, and write it back. Chapter 6, “Modifying Data Through SQL Programs,” discusses programs that modify data.

Using a Cursor: A Parts Explosion

When you use a cursor, supplemented by program logic, you can solve problems that plain SQL cannot solve. One of these problems is the parts-explosion problem, sometimes called bill-of-materials processing. At the heart of this problem is a recursive relationship among objects; one object contains other objects, which contain yet others.

The problem is usually stated in terms of a manufacturing inventory. A company makes a variety of parts, for example. Some parts are discrete, but some are assemblages of other parts.

These relationships are documented in a single table, which might be called **contains**. The column **contains.parent** holds the part numbers of parts that are assemblages. The column **contains.child** has the part number of a part that is a component of the parent. If part number 123400 is an assembly of nine parts, nine rows exist with 123400 in the first column and other part numbers in the second. Figure 5-5 shows one of the rows that describe part number 123400.



CONTAINS

PARENT	CHILD
FK NN	FK NN
123400 432100	432100 765899

Figure 5-5
Parts-Explosion Problem

Here is the parts-explosion problem: given a part number, produce a list of all parts that are components of that part. The following example is a sketch of one solution, as implemented in INFORMIX-ESQL/C:

```
int part_list[200];

boom(top_part)
int top_part;
{
    long this_part, child_part;
    int next_to_do = 0, next_free = 1;
    part_list[next_to_do] = top_part;

    EXEC SQL DECLARE part_scan CURSOR FOR
        SELECT child INTO child_part FROM contains
            WHERE parent = this_part;
    while(next_to_do < next_free)
    {
        this_part = part_list[next_to_do];
        EXEC SQL OPEN part_scan;
        while(SQLCODE == 0)
        {
            EXEC SQL FETCH part_scan;
            if(SQLCODE == 0)
            {
                part_list[next_free] = child_part;
                next_free += 1;
            }
        }
        EXEC SQL CLOSE part_scan;
        next_to_do += 1;
    }
    return (next_free - 1);
}
```

Technically speaking, each row of the **contains** table is the head node of a directed acyclic graph, or *tree*. The function performs a breadth-first search of the tree whose root is the part number passed as its parameter. The function uses a cursor named **part_scan** to return all the rows with a particular value in the **parent** column. The innermost `while` loop opens the **part_scan** cursor, fetches each row in the selection set, and closes the cursor when the part number of each component has been retrieved.

This function addresses the heart of the parts-explosion problem, but the function is not a complete solution. For example, it does not allow for components that appear at more than one level in the tree. Furthermore, a practical **contains** table would also have a column **count**, giving the count of **child** parts used in each **parent**. A program that returns a total count of each component part is much more complicated.

The iterative approach described earlier is not the only way to approach the parts-explosion problem. If the number of generations has a fixed limit, you can solve the problem with a single SELECT statement using nested, outer self-joins.

If up to four generations of parts can be contained within one top-level part, the following SELECT statement returns all of them:

```
SELECT a.parent, a.child, b.child, c.child, d.child
    FROM contains a
        OUTER (contains b,
            OUTER (contains c, outer contains d))
    WHERE a.parent = top_part_number
        AND a.child = b.parent
        AND b.child = c.parent
        AND c.child = d.parent
```

This SELECT statement returns one row for each line of descent rooted in the part given as **top_part_number**. Null values are returned for levels that do not exist. (Use indicator variables to detect them.) To extend this solution to more levels, select additional nested outer joins of the **contains** table.You can also revise this solution to return counts of the number of parts at each level.

Dynamic SQL

Although static SQL is useful, it requires that you know the exact content of every SQL statement at the time you write the program. For example, you must state exactly which columns are tested in any WHERE clause and exactly which columns are named in any select list.

No problem exists when you write a program to perform a well-defined task. But the database tasks of some programs cannot be perfectly defined in advance. In particular, a program that must respond to an interactive user might need to compose SQL statements in response to what the user enters.

Dynamic SQL allows a program to form an SQL statement during execution, so that user input determines the contents of the statement. This action is performed in the following steps:

1. The program assembles the text of an SQL statement as a character string, which is stored in a program variable.
2. It executes a PREPARE statement, which asks the database server to examine the statement text and prepare it for execution.
3. It uses the EXECUTE statement to execute the prepared statement.

In this way, a program can construct and then use any SQL statement, based on user input of any kind. For example, it can read a file of SQL statements and prepare and execute each one.

DB-Access, a utility that you can use to explore SQL interactively, is an INFORMIX-ESQL/C program that constructs, prepares, and executes SQL statements dynamically. For example, DB-Access lets you use simple, interactive menus to specify the columns of a table. When you are finished, DB-Access builds the necessary CREATE TABLE or ALTER TABLE statement dynamically and prepares and executes it.

Preparing a Statement

In form, a dynamic SQL statement is like any other SQL statement that is written into a program, except that it cannot contain the names of any host variables.

A dynamic SQL statement has two restrictions. First, if it is a SELECT statement, it cannot include the INTO clause. The INTO clause names host variables into which column data is placed, and host variables are not allowed in a dynamic statement. Second, wherever the name of a host variable normally appears in an expression, a question mark (?) is written as a placeholder.

You can prepare a statement in this form for execution with the PREPARE statement. The following example is written in INFORMIX-ESQL/C:

```
EXEC SQL prepare query_2 from
        'select * from orders
            where customer_num = ? and
                order_date > ?';
```

The two question marks in this example indicate that when the statement is executed, the values of host variables are used at those two points.

You can prepare almost any SQL statement dynamically. The only statements that you cannot prepare are the ones directly concerned with dynamic SQL and cursor management, such as the PREPARE and OPEN statements. After you prepare an UPDATE or DELETE statement, it is a good idea to test the fifth field of SQLWARN to see if you used a WHERE clause (see "SQLWARN Array" on page 5-11).

The result of preparing a statement is a data structure that represents the statement. This data structure is not the same as the string of characters that produced it. In the PREPARE statement, you give a name to the data structure; it is **query_2** in the preceding example. This name is used to execute the prepared SQL statement.

The PREPARE statement does not limit the character string to one statement. It can contain multiple SQL statements, separated by semicolons. The following example shows a fairly complex transaction in INFORMIX-ESQL/C:

```
strcpy(big_query, "UPDATE account SET balance = balance + ?
WHERE customer_id = ?; \ UPDATE teller SET balance =
balance + ? WHERE teller_id = ?;");
EXEC SQL PREPARE big1 FROM :big_query;
```

When this list of statements is executed, host variables must provide values for six place-holding question marks. Although it is more complicated to set up a multistatement list, performance is often better because fewer exchanges take place between the program and the database server.

Executing Prepared SQL

Once a statement is prepared, it can be executed multiple times. Statements other than SELECT statements, and SELECT statements that return only a single row, are executed with the EXECUTE statement.

The following INFORMIX-ESQL/C code prepares and executes a multistatement update of a bank account:

```
EXEC SQL BEGIN DECLARE SECTION;
char bigquery[270] = "begin work;";
EXEC SQL END DECLARE SECTION;
stcat ("update account set balance = balance + ? where ", bigquery);
stcat ("acct_number = ?;', bigquery);
stcat ("update teller set balance = balance + ? where ", bigquery);
stcat ("teller_number = ?;', bigquery);
stcat ("update branch set balance = balance + ? where ", bigquery);
stcat ("branch_number = ?;', bigquery);
stcat ("insert into history values(timestamp, values);", bigquery);

EXEC SQL prepare bigq from :bigquery;

EXEC SQL execute bigq using :delta, :acct_number, :delta,
    :teller_number, :delta, :branch_number;

EXEC SQL commit work;
```

The USING clause of the EXECUTE statement supplies a list of host variables whose values are to take the place of the question marks in the prepared statement. If a SELECT (or an EXECUTE PROCEDURE) returns only one row, you can use the INTO clause of EXECUTE to specify the host variables that receive the values.

Dynamic Host Variables

SQL APIs, which support dynamically allocated data objects, take dynamic statements one step further. They let you dynamically allocate the host variables that receive column data.

Dynamic allocation of variables makes it possible to take an arbitrary SELECT statement from program input, determine how many values it produces and their data types, and allocate the host variables of the appropriate types to hold them.

The key to this ability is the DESCRIBE statement. It takes the name of a prepared SQL statement and returns information about the statement and its contents. It sets SQLCODE to specify the type of statement; that is, the verb with which it begins. If the prepared statement is a SELECT statement, the DESCRIBE statement also returns information about the selected output data. If the prepared statement is an INSERT statement, the DESCRIBE statement returns information about the input parameters. The data structure to which a DESCRIBE statement returns information is a predefined data structure that is allocated for this purpose and is known as a system-descriptor area. If you are using INFORMIX-ESQL/C, you can use a system-descriptor area or, as an alternative, an **sqlda** structure.

The data structure that a DESCRIBE statement returns or references for a SELECT statement includes an array of structures. Each structure describes the data that is returned for one item in the select list. The program can examine the array and discover that a row of data includes a decimal value, a character value of a certain length, and an integer.

With this information, the program can allocate memory to hold the retrieved values and put the necessary pointers in the data structure for the database server to use.

Freeing Prepared Statements

A prepared SQL statement occupies space in memory. With some database servers, it can consume space owned by the database server as well as space that belongs to the program. This space is released when the program terminates, but in general, you should free this space when you finish with it.

You can use the FREE statement to release this space. The FREE statement takes either the name of a statement or the name of a cursor that was declared for a statement name, and releases the space allocated to the prepared statement. If more than one cursor is defined on the statement, freeing the statement does not free the cursor.

Quick Execution

For simple statements that do not require a cursor or host variables, you can combine the actions of the PREPARE, EXECUTE, and FREE statements into a single operation. The following example shows how the EXECUTE IMMEDIATE statement takes a character string, prepares it, executes it, and frees the storage in one operation:

```
   EXEC SQL execute immediate 'drop index my_temp_index';
```

This capability makes it easy to write simple SQL operations. However, because no USING clause is allowed, the EXECUTE IMMEDIATE statement cannot be used for SELECT statements.

Embedding Data-Definition Statements

Data-definition statements, the SQL statements that create databases and modify the definitions of tables, are not usually put into programs. The reason is that they are rarely performed. A database is created once, but it is queried and updated many times.

The creation of a database and its tables is generally done interactively, using DB-Access or Relational Object Manager. These tools can also be run from a file of statements, so that the creation of a database can be done with one operating-system command. The data definition statements are documented in the *Informix Guide to SQL: Syntax* and the *Informix Guide to Database Design and Implementation*.

Embedding Grant and Revoke Privileges

One task related to data definition is performed repeatedly: granting and revoking privileges. Because privileges must be granted and revoked frequently, and possibly by users who are not skilled in SQL, it can be useful to package the GRANT and REVOKE statements in programs to give them a simpler, more convenient user interface.

The GRANT and REVOKE statements are especially good candidates for dynamic SQL. Each statement takes the following parameters:

- A list of one or more privileges
- A table name
- The name of a user

You probably need to supply at least some of these values based on program input (from the user, command-line parameters, or a file) but none can be supplied in the form of a host variable. The syntax of these statements does not allow host variables at any point.

The only alternative is to assemble the parts of a statement into a character string and to prepare and execute the assembled statement. Program input can be incorporated into the prepared statement as characters.

The following INFORMIX-ESQL/C function assembles a GRANT statement from parameters, and then prepares and executes it:

```
char priv_to_grant[100];
char table_name[20];
char user_id[20];

table_grant(priv_to_grant, table_name, user_id)
char *priv_to_grant;
char *table_name;
char *user_id;
{
    EXEC SQL BEGIN DECLARE SECTION;
    char grant_stmt[200];
    EXEC SQL END DECLARE SECTION;

    sprintf(grant_stmt, " GRANT %s ON %s TO %s",
        priv_to_grant, table_name, user_id);
    PREPARE the_grant FROM :grant_stmt;
    if(SQLCODE == 0)
        EXEC SQL EXECUTE the_grant;
    else
        printf("Sorry, got error # %d attempting %s",
            SQLCODE, grant_stmt);

    EXEC SQL FREE the_grant;
}
```

The opening statement of the function, shown in the following example, specifies its name and its three parameters. The three parameters specify the privileges to grant, the name of the table on which to grant privileges, and the ID of the user to receive them:

```
table_grant(priv_to_grant, table_name, user_id)
char *priv_to_grant;
char *table_name;
char *user_id;
```

The function uses the statements in the following example to define a local variable, **grant_stmt**, which is used to assemble and hold the GRANT statement:

```
EXEC SQL BEGIN DECLARE SECTION;
    char grant_stmt[200];
EXEC SQL END DECLARE SECTION;
```

As the following example illustrates, the GRANT statement is created by concatenating the constant parts of the statement and the function parameters:

```
sprintf(grant_stmt, " GRANT %s ON %s TO %s",priv_to_grant, table_name, user_id);
```

This statement concatenates the following six character strings:

- 'GRANT'
- The parameter that specifies the privileges to be granted
- 'ON'
- The parameter that specifies the table name
- 'TO'
- The parameter that specifies the user

The result is a complete GRANT statement composed partly of program input. The PREPARE statement passes the assembled statement text to the database server for parsing.

If the database server returns an error code in SQLCODE following the PREPARE statement, the function displays an error message. If the database server approves the form of the statement, it sets a zero return code. This action does not guarantee that the statement is executed properly; it means only that the statement has correct syntax. It might refer to a nonexistent table or contain many other kinds of errors that can be detected only during execution. The following portion of the example checks that **the_grant** was prepared successfully before executing it:

```
if(SQLCODE == 0)
    EXEC SQL EXECUTE the_grant;
else
    printf("Sorry, got error # %d attempting %s", SQLCODE, grant_stmt);
```

If the preparation is successful, `SQLCODE = = 0`, the next step executes the prepared statement.

Summary

SQL statements can be written into programs as if they were normal statements of the programming language. Program variables can be used in WHERE clauses, and data from the database can be fetched into them. A preprocessor translates the SQL code into procedure calls and data structures.

Statements that do not return data, or queries that return only one row of data, are written like ordinary imperative statements of the language. Queries that can return more than one row are associated with a cursor that represents the current row of data. Through the cursor, the program can fetch each row of data as it is needed.

Static SQL statements are written into the text of the program. However, the program can form new SQL statements dynamically, as it runs, and execute them also. In the most advanced cases, the program can obtain information about the number and types of columns that a query returns and dynamically allocate the memory space to hold them.

Chapter

6 Modifying Data Through SQL Programs

Using DELETE . . . 6-3
    Direct Deletions . . . 6-4
        Errors During Direct Deletions . . . 6-4
        Using Transaction Logging . . . 6-5
        Coordinated Deletions . . . 6-6
    Deleting with a Cursor . . . 6-7

Using INSERT . . . 6-9
    Using an Insert Cursor . . . 6-9
        Declaring an Insert Cursor . . . 6-9
        Inserting with a Cursor. . . . 6-10
        Status Codes After PUT and FLUSH . . . 6-11
    Rows of Constants. . . . 6-12
    An Insert Example. . . . 6-12

Using UPDATE . . . 6-15
    Using an Update Cursor . . . 6-15
        The Purpose of the Keyword UPDATE . . . 6-16
        Updating Specific Columns . . . 6-16
        UPDATE Keyword Not Always Needed. . . . 6-16
    Cleaning Up a Table . . . 6-17

Summary . . . 6-18

The preceding chapter introduced the idea of inserting or embedding SQL statements, especially the SELECT statement, into programs written in other languages. Embedded SQL enables a program to retrieve rows of data from a database.

This chapter discusses the issues that arise when a program needs to delete, insert, or update rows to modify the database. As in Chapter 5, "Programming with SQL," this chapter prepares you for reading the manual for your Informix embedded language.

The general use of the INSERT, UPDATE, and DELETE statements is discussed in Chapter 4, "Modifying Data." This chapter examines their use from within a program. You can easily embed the statements in a program, but it can be difficult to handle errors and to deal with concurrent modifications from multiple programs.

Using DELETE

To delete rows from a table, a program executes a DELETE statement. The DELETE statement can specify rows in the usual way with a WHERE clause, or it can refer to a single row, the last one fetched through a specified cursor.

Whenever you delete rows, you must consider whether rows in other tables depend on the deleted rows. This problem of coordinated deletions is covered in Chapter 4, "Modifying Data." The problem is the same when deletions are made from within a program.

Direct Deletions

You can embed a DELETE statement in a program. The following example uses INFORMIX-ESQL/C:

```
EXEC SQL delete from items
    where order_num = :onum;
```

You can also prepare and execute a statement of the same form dynamically. In either case, the statement works directly on the database to affect one or more rows.

The WHERE clause in the example uses the value of a host variable named **onum**. Following the operation, results are posted in SQLSTATE and in the **sqlca** structure, as usual. The third element of the SQLERRD array contains the count of rows deleted even if an error occurs. The value in SQLCODE shows the overall success of the operation. If the value is not negative, no errors occurred and the third element of SQLERRD is the count of all rows that satisfied the WHERE clause and were deleted.

Errors During Direct Deletions

When an error occurs, the statement ends prematurely. The values in SQLSTATE and in SQLCODE and the second element of SQLERRD explain its cause, and the count of rows reveals how many rows were deleted. For many errors, that count is zero because the errors prevented the database server from beginning the operation. For example, if the named table does not exist, or if a column tested in the WHERE clause is renamed, no deletions are attempted.

However, certain errors can be discovered after the operation begins and some rows are processed. The most common of these errors is a lock conflict. The database server must obtain an exclusive lock on a row before it can delete that row. Other programs might be using the rows from the table, preventing the database server from locking a row. Because the issue of locking affects all types of modifications, it is discussed in Chapter 7, “Programming for a Multiuser Environment.”

Other, rarer types of errors can strike after deletions begin. For example, hardware errors that occur while the database is being updated.

Using Transaction Logging

The best way to prepare for any kind of error during a modification is to use transaction logging. In the event of an error, you can tell the database server to put the database back the way it was. The following example is based on the example in the section "Direct Deletions" on page 6-4, which is extended to use transactions:

```
EXEC SQL begin work;/* start the transaction*/
EXEC SQL delete from items
      where order_num = :onum;
del_result = sqlca.sqlcode;/* save two error */
del_isamno = sqlca.sqlerrd[1];/* ...code numbers */
del_rowcnt = sqlca.sqlerrd[2];/* ...and count of rows */
if (del_result < 0)/* some problem, */
    EXEC SQL rollback work;/* ...put everything back */
else    /* everything worked OK, */
    EXEC SQL commit work;/* ...finish transaction */
```

A key point in this example is that the program saves the important return values in the **sqlca** structure before it ends the transaction. Both the ROLLBACK WORK and COMMIT WORK statements, like other SQL statements, set return codes in the **sqlca** structure. Executing a ROLLBACK WORK statement after an error wipes out the error code; unless it was saved, it cannot be reported to the user.

The advantage of using transactions is that the database is left in a known, predictable state no matter what goes wrong. No question remains about how much of the modification is completed; either all of it or none of it is completed.

In a database with logging, if a user does not start an explicit transaction, the database server initiates an internal transaction prior to execution of the statement and terminates the transaction after execution completes or fails. If the statement execution succeeds, the internal transaction is committed. If the statement fails, the internal transaction is rolled back.

Coordinated Deletions

The usefulness of transaction logging is particularly clear when you must modify more than one table. For example, consider the problem of deleting an order from the demonstration database. In the simplest form of the problem, you must delete rows from two tables, **orders** and **items**, as the following example of INFORMIX-ESQL/C shows:

```
EXEC SQL BEGIN WORK;
EXEC SQL DELETE FROM items
    WHERE order_num = :o_num;
if (SQLCODE >= 0)
{
    EXEC SQL DELETE FROM orders
        WHERE order_num = :o_num;
    if (SQLCODE >= 0)
        EXEC SQL COMMIT WORK;
    else
    {
        printf("Error %d on DELETE", SQLCODE);
        EXEC SQL ROLLBACK WORK;
    }
}
```

The logic of this program is much the same whether or not transactions are used. If they are not used, the person who sees the error message has a much more difficult set of decisions to make. Depending on when the error occurred, one of the following situations applies:

- No deletions were performed; all rows with this order number remain in the database.
- Some, but not all, item rows were deleted; an order record with only some items remains.
- All item rows were deleted, but the order row remains.
- All rows were deleted.

In the second and third cases, the database is corrupted to some extent; it contains partial information that can cause some queries to produce wrong answers. You must take careful action to restore consistency to the information. When transactions are used, all these uncertainties are prevented.

Deleting with a Cursor

You can also write a DELETE statement with a cursor to delete the row that was last fetched. Deleting rows in this manner lets you program deletions based on conditions that cannot be tested in a WHERE clause, as the following example shows. The following example applies only to databases that are not ANSI compliant because of the way that the beginning and end of the transaction are set up.

Warning: *The design of the ESQL/C function in this example is unsafe. It depends on the current isolation level for correct operation. Isolation levels are discussed later in the chapter. For information on isolation levels see Chapter 7, "Programming for a Multiuser Environment." Even when the function works as intended, its effects depend on the physical order of rows in the table, which is not generally a good idea.*

```
int delDupOrder()
{
    int ord_num;
    int dup_cnt, ret_code;

    EXEC SQL declare scan_ord cursor for
        select order_num, order_date
            into :ord_num, :ord_date
            from orders for update;
    EXEC SQL open scan_ord;
    if (sqlca.sqlcode != 0)
        return (sqlca.sqlcode);
    EXEC SQL begin work;
    for(;;)
    {
        EXEC SQL fetch next scan_ord;
        if (sqlca.sqlcode != 0) break;
        dup_cnt = 0; /* default in case of error */
        EXEC SQL select count(*) into dup_cnt from orders
            where order_num = :ord_num;
        if (dup_cnt > 1)
        {
            EXEC SQL delete from orders
                where current of scan_ord;
            if (sqlca.sqlcode != 0)
                break;
        }
    }
    ret_code = sqlca.sqlcode;
    if (ret_code == 100) /* merely end of data */
        EXEC SQL commit work;
    else    /* error on fetch or on delete */
        EXEC SQL rollback work;
    return (ret_code);
}
```

The purpose of the function is to delete rows that contain duplicate order numbers. In fact, in the demonstration database, the **orders.order_num** column has a unique index, so duplicate rows cannot occur in it. However, a similar function can be written for another database; this one uses familiar column names.

The function declares **scan_ord**, a cursor to scan all rows in the **orders** table. It is declared with the FOR UPDATE clause, which states that the cursor can modify data. If the cursor opens properly, the function begins a transaction and then loops over rows of the table. For each row, it uses an embedded SELECT statement to determine how many rows of the table have the order number of the current row. (This step fails without the correct isolation level, as described in Chapter 7, "Programming for a Multiuser Environment.")

In the demonstration database, with its unique index on this table, the count returned to **dup_cnt** is always one. However, if it is greater, the function deletes the current row of the table, reducing the count of duplicates by one.

Clean-up functions of this sort are sometimes needed, but they generally need more sophisticated design. This function deletes all duplicate rows except the last one that the database server returns. That order has nothing to do with the contents of the rows or their meanings. You can improve the function in the previous example by adding, perhaps, an ORDER BY clause to the cursor declaration. However, you cannot use ORDER BY and FOR UPDATE together. A better approach is presented in "An Insert Example" on page 6-12.

Using INSERT

You can embed the INSERT statement in programs. Its form and use in a program are the same as described in Chapter 4, “Modifying Data,” with the additional feature that you can use host variables in expressions, both in the VALUES and WHERE clauses. Moreover, a program has the additional ability to insert rows with a cursor.

Using an Insert Cursor

The DECLARE CURSOR statement has many variations. Most are used to create cursors for different kinds of scans over data, but one variation creates a special kind of cursor called an *insert cursor*. You use an insert cursor with the PUT and FLUSH statements to insert rows into a table in bulk efficiently.

Declaring an Insert Cursor

To create an insert cursor, declare a cursor to be for an INSERT statement instead of a SELECT statement. You cannot use such a cursor to fetch rows of data; you can use it only to insert them. The following code example shows the declaration of an insert cursor:

```
DEFINE the_company LIKE customer.company,
    the_fname LIKE customer.fname,
    the_lname LIKE customer.lname
DECLARE new_custs CURSOR FOR
    INSERT INTO customer (company, fname, lname)
        VALUES (the_company, the_fname, the_lname)
```

When you open an insert cursor, a buffer is created in memory to hold a block of rows. The buffer receives rows of data as the program produces them; then they are passed to the database server in a block when the buffer is full. This reduces the amount of communication between the program and the database server, and it lets the database server insert the rows with less difficulty. As a result, the insertions go faster.

The buffer is always made large enough to hold at least two rows of inserted values. It is large enough to hold more than two rows when the rows are shorter than the minimum buffer size.

Inserting with a Cursor

The code in the previous example prepares an insert cursor for use. The continuation, as the following example shows, demonstrates how the cursor can be used. For simplicity, this example assumes that a function named **next_cust** returns either information about a new customer or null data to signal the end of input.

```
EXEC SQL BEGIN WORK;
EXEC SQL OPEN new_custs;
while(SQLCODE == 0)
{
    next_cust();
    if(the_company == NULL)
        break;
    EXEC SQL PUT new_custs;
}
if(SQLCODE == 0) /* if no problem with PUT */
{
    EXEC SQL FLUSH new_custs;/* write any rows left */
    if(SQLCODE == 0)/* if no problem with FLUSH */
        EXEC SQL COMMIT WORK;/* commit changes */
}
else
    EXEC SQL ROLLBACK WORK;/* else undo changes */
```

The code in this example calls **next_cust** repeatedly. When it returns non-null data, the PUT statement sends the returned data to the row buffer. When the buffer fills, the rows it contains are automatically sent to the database server. The loop normally ends when **next_cust** has no more data to return. Then the FLUSH statement writes any rows that remain in the buffer, after which the transaction terminates.

Examine the INSERT statement on page 6-9 once more. The statement by itself, not part of a cursor definition, inserts a single row into the **customer** table. In fact, the whole apparatus of the insert cursor can be dropped from the example code, and the INSERT statement can be written into the code where the PUT statement now stands. The difference is that an insert cursor causes a program to run somewhat faster.

Status Codes After PUT and FLUSH

When a program executes a PUT statement, the program should test whether the row is placed in the buffer successfully. If the new row fits in the buffer, the only action of PUT is to copy the row to the buffer. No errors can occur in this case. However, if the row does not fit, the entire buffer load is passed to the database server for insertion, and an error can occur.

The values returned into the SQL Communications Area (SQLCA) give the program the information it needs to sort out each case. SQLCODE and SQLSTATE are set after every PUT statement, to zero if no error occurs and to a negative error code if an error occurs.

The third element of SQLERRD is set to the number of rows actually inserted into the table. It is set to zero if the new row is merely moved to the buffer; to the count of rows that are in the buffer if the buffer load is inserted without error; or to the count of rows inserted before an error occurs, if one does occur.

Read the code once again to see how SQLCODE is used (see the previous example). First, if the OPEN statement yields an error, the loop is not executed because the WHILE condition fails, the FLUSH operation is not performed, and the transaction rolls back.Second, if the PUT statement returns an error, the loop ends because of the WHILE condition, the FLUSH operation is not performed, and the transaction rolls back. This condition can occur only if the loop generates enough rows to fill the buffer at least once; otherwise, the PUT statement cannot generate an error.

The program might end the loop with rows still in the buffer, possibly without inserting any rows. At this point, the SQL status is zero, and the FLUSH operation occurs. If the FLUSH operation produces an error code, the transaction rolls back. Only when all inserts are successfully performed is the transaction committed.

Rows of Constants

The insert cursor mechanism supports one special case where high performance is easy to obtain. In this case, all the values listed in the INSERT statement are constants: no expressions and no host variables are listed, just literal numbers and strings of characters. No matter how many times such an INSERT operation occurs, the rows it produces are identical. When the rows are identical, copying, buffering, and transmitting each identical row is pointless.

Instead, for this kind of INSERT operation, the PUT statement does nothing except to increment a counter. When a FLUSH operation is finally performed, a single copy of the row, and the count of inserts, is passed to the database server. The database server creates and inserts that many rows in one operation.

You do not usually insert a quantity of identical rows. You can insert identical rows when you first establish a database, to populate a large table with null data.

An Insert Example

"Deleting with a Cursor" on page 6-7 contains an example of the DELETE statement whose purpose is to look for and delete duplicate rows of a table. A better way to perform this task is to select the desired rows instead of deleting the undesired ones. The code in the following INFORMIX-ESQL/C example shows one way to do this.

```
EXEC SQL BEGIN DECLARE SECTION;
    long last_ord = 1;
    struct {
        long int o_num;
        date     o_date;
        long     c_num;
        char     o_shipinst[40];
        char     o_backlog;
        char     o_po[10];
        date     o_shipdate;
        decimal  o_shipwt;
        decimal  o_shipchg;
        date     o_paiddate;
        } ord_row;
EXEC SQL END DECLARE SECTION;

EXEC SQL BEGIN WORK;
```

```
    EXEC SQL INSERT INTO new_orders
        SELECT * FROM orders main
            WHERE 1 = (SELECT COUNT(*) FROM orders minor
                WHERE main.order_num = minor.order_num);
    EXEC SQL COMMIT WORK;

    EXEC SQL DECLARE dup_row CURSOR FOR
        SELECT * FROM orders main INTO :ord_row
            WHERE 1 < (SELECT COUNT(*) FROM orders minor
                WHERE main.order_num = minor.order_num)
            ORDER BY order_date;
    EXEC SQL DECLARE ins_row CURSOR FOR
        INSERT INTO new_orders VALUES (:ord_row);

    EXEC SQL BEGIN WORK;
    EXEC SQL OPEN ins_row;
    EXEC SQL OPEN dup_row;
    while(SQLCODE == 0)
    {
        EXEC SQL FETCH dup_row;
        if(SQLCODE == 0)
        {
            if(ord_row.o_num != last_ord)
                EXEC SQL PUT ins_row;
            last_ord = ord_row.o_num
            continue;
        }
        break;
    }
    if(SQLCODE != 0 && SQLCODE != 100)
        EXEC SQL ROLLBACK WORK;
    else
        EXEC SQL COMMIT WORK;
    EXEC SQL CLOSE ins_row;
    EXEC SQL CLOSE dup_row;
```

This example begins with an ordinary INSERT statement, which finds all the nonduplicated rows of the table and inserts them into another table, presumably created before the program started. That action leaves only the duplicate rows. (In the demonstration database, the **orders** table has a unique index and cannot have duplicate rows. Assume that this example deals with some other database.)

The code in the previous example then declares two cursors. The first, called **dup_row**, returns the duplicate rows in the table. Because **dup_row** is for input only, it can use the ORDER BY clause to impose some order on the duplicates other than the physical record order used in the example on page 6-7. In this example, the duplicate rows are ordered by their dates (the oldest one remains), but you can use any other order based on the data.

The second cursor, **ins_row**, is an insert cursor. This cursor takes advantage of the ability to use a C structure, **ord_row**, to supply values for all columns in the row.

The remainder of the code examines the rows that are returned through **dup_row**. It inserts the first one from each group of duplicates into the new table and disregards the rest.

For the sake of brevity, the preceding example uses the simplest kind of error handling. If an error occurs before all rows have been processed, the sample code rolls back the active transaction.

How Many Rows Were Affected?

When your program uses a cursor to select rows, it can test SQLCODE for `100` (or SQLSTATE for `02000`), the end-of-data return code. This code is set to indicate that no rows, or no more rows, satisfy the query conditions. For databases that are not ANSI compliant, the end-of-data return code is set in SQLCODE or SQLSTATE only following SELECT statements; it is not used following DELETE, INSERT, or UPDATE statements. For ANSI-compliant databases, SQLCODE is also set to `100` for updates, deletes, and inserts that affect zero rows.

A query that finds no data is not a success. However, an UPDATE or DELETE statement that happens to update or delete no rows is still considered a success. It updated or deleted the set of rows that its WHERE clause said it should; however, the set was empty.

In the same way, the INSERT statement does not set the end-of-data return code even when the source of the inserted rows is a SELECT statement, and the SELECT statement selected no rows. The INSERT statement is a success because it inserted as many rows as it was asked to (that is, zero).

To find out how many rows are inserted, updated, or deleted, a program can test the third element of SQLERRD. The count of rows is there, regardless of the value (zero or negative) in SQLCODE.

Using UPDATE

You can embed the UPDATE statement in a program in any of the forms described in Chapter 4, “Modifying Data,” with the additional feature that you can name host variables in expressions, both in the SET and WHERE clauses. Moreover, a program can update the row that is addressed by a cursor.

Using an Update Cursor

An *update cursor* permits you to delete or update the current row; that is, the most recently fetched row. The following example in INFORMIX-ESQL/C shows the declaration of an update cursor:

```
EXEC SQL
    DECLARE names CURSOR FOR
        SELECT fname, lname, company
        FROM customer
    FOR UPDATE;
```

The program that uses this cursor can fetch rows in the usual way.

```
EXEC SQL
    FETCH names INTO :FNAME, :LNAME, :COMPANY;
```

If the program then decides that the row needs to be changed, it can do so.

```
if (strcmp(COMPANY, "SONY") ==0)
    {
    EXEC SQL
        UPDATE customer
            SET fname = 'Midori', lname = 'Tokugawa'
            WHERE CURRENT OF names;
    }
```

The words `CURRENT OF names` take the place of the usual test expressions in the WHERE clause. In other respects, the UPDATE statement is the same as usual, even including the specification of the table name, which is implicit in the cursor name but still required.

The Purpose of the Keyword UPDATE

The purpose of the keyword UPDATE in a cursor is to let the database server know that the program can update (or delete) any row that it fetches. The database server places a more demanding lock on rows that are fetched through an update cursor and a less demanding lock when it fetches a row for a cursor that is not declared with that keyword. This action results in better performance for ordinary cursors and a higher level of concurrent use in a multiprocessing system. (Levels of locks and concurrent use are discussed in Chapter 7, “Programming for a Multiuser Environment.”)

Updating Specific Columns

The following example has updated specific columns of the preceding example of an update cursor:

```
EXEC SQL
    DECLARE names CURSOR FOR
        SELECT fname, lname, company, phone
            INTO  :FNAME,:LNAME,:COMPANY,:PHONE FROM customer
    FOR UPDATE OF fname, lname
END-EXEC.
```

Only the **fname** and **lname** columns can be updated through this cursor. A statement such as the following one is rejected as an error:

```
EXEC SQL
    UPDATE customer
        SET company = 'Siemens'
        WHERE CURRENT OF names
END-EXEC.
```

If the program attempts such an update, an error code is returned and no update occurs. An attempt to delete with WHERE CURRENT OF is also rejected because deletion affects all columns.

UPDATE Keyword Not Always Needed

The ANSI standard for SQL does not provide for the FOR UPDATE clause in a cursor definition. When a program uses an ANSI-compliant database, it can update or delete with any cursor.

Cleaning Up a Table

A final, hypothetical example of how to use an update cursor presents a problem that should never arise with an established database but could arise in the initial design phases of an application.

In the example, a large table named **target** is created and populated. A character column, **datcol**, inadvertently acquires some null values. These rows should be deleted. Furthermore, a new column, **serials**, is added to the table with the ALTER TABLE statement. This column is to have unique integer values installed. The following example shows the INFORMIX-ESQL/C code you use to accomplish these tasks:

```
EXEC SQL BEGIN DECLARE SECTION;
char dcol[80];
short dcolint;
int sequence;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE target_row CURSOR FOR
    SELECT datcol
        INTO :dcol:dcolint
        FROM target
    FOR UPDATE OF serials;
EXEC SQL BEGIN WORK;
EXEC SQL OPEN target_row;
if (sqlca.sqlcode == 0) EXEC SQL FETCH NEXT target_row;
for(sequence = 1; sqlca.sqlcode == 0; ++sequence)
{
    if (dcolint < 0) /* null datcol */
        EXEC SQL DELETE WHERE CURRENT OF target_row;
    else
        EXEC SQL UPDATE target SET serials = :sequence
            WHERE CURRENT OF target_row;
}
if (sqlca.sqlcode >= 0)
   EXEC SQL COMMIT WORK;
else EXEC SQL ROLLBACK WORK;
```

Summary

A program can execute the INSERT, DELETE, and UPDATE statements as described in Chapter 4, “Modifying Data.” A program also can scan through a table with a cursor, updating or deleting selected rows. It can also use a cursor to insert rows, with the benefit that the rows are buffered and sent to the database server in blocks.

In all these activities, you must make sure that the program detects errors and returns the database to a known state when an error occurs. The most important tool for doing this is the transaction. Without transaction logging, it is more difficult to write programs that can recover from errors.

Chapter

7

Programming for a Multiuser Environment

Concurrency and Performance 7-3

Locking and Integrity 7-3

Locking and Performance 7-4

Concurrency Issues. 7-4

How Locks Work 7-6
Kinds of Locks 7-7
Lock Scope 7-7
Database Locks 7-8
Table Locks 7-8
TABLE Lock Mode for Informix Dynamic Server with AD and XP Options 7-9
Page, Row, and Key Locks. 7-10
Duration of a Lock. 7-11
Locks While Modifying 7-11

Setting the Isolation Level 7-12
Comparing SET TRANSACTION with SET ISOLATION 7-12
ANSI Read Uncommitted and Informix Dirty Read Isolation . . . 7-14
ANSI Read Committed and Informix Committed Read Isolation. . 7-14
Informix Cursor Stability Isolation 7-15
ANSI Serializable, ANSI Repeatable Read, and Informix Repeatable Read Isolation. 7-17

Controlling Data Modification with Access Modes 7-18

Setting the Lock Mode 7-19
    Waiting for Locks 7-19
    Not Waiting for Locks 7-19
    Waiting a Limited Time 7-20
    Handling a Deadlock 7-20
    Handling External Deadlock 7-21

Simple Concurrency 7-21

Hold Cursors 7-21

Summary 7-23

If your database is contained in a single-user workstation and is not connected on a network to other computers, your programs can modify data freely. In all other cases, you must allow for the possibility that, while your program is modifying data, another program is reading or modifying the same data. This situation describes *concurrency*: two or more independent uses of the same data at the same time. This chapter addresses concurrency, locking, and isolation levels.

Concurrency and Performance

Concurrency is crucial to good performance in a multiprogramming system. When access to the data is *serialized* so that only one program at a time can use it, processing slows dramatically.

Locking and Integrity

Unless controls are placed on the use of data, concurrency can lead to a variety of negative effects. Programs can read obsolete data, or modifications can be lost even though they were apparently completed.

To prevent errors of this kind, the database server imposes a system of *locks*. A lock is a claim, or reservation, that a program can place on a piece of data. The database server guarantees that, as long as the data is locked, no other program can modify it. When another program requests the data, the database server either makes the program wait or turns it back with an error.

Locking and Performance

Because a lock serializes access to one piece of data, it reduces concurrency; any other programs that want access to that data must wait. The database server can place a lock on a single row, a disk page (which holds multiple rows), a whole table, or an entire database. The more locks it places and the larger the objects it locks, the more concurrency is reduced. The fewer the locks and the smaller the locked objects, the greater concurrency and performance can be.

The following sections discuss how a program can achieve the following goals:

- Place all the locks necessary to ensure data integrity
- Lock the fewest, smallest pieces of data possible consistent with the preceding goal

Concurrency Issues

To understand the hazards of concurrency, you must think in terms of multiple programs, each executing at its own speed. Suppose that your program is fetching rows through the following cursor:

```
EXEC SQL DECLARE sto_curse CURSOR FOR
    SELECT * FROM stock
        WHERE manu_code = 'ANZ';
```

The transfer of each row from the database server to the program takes time. During and between transfers, other programs can perform other database operations. At about the same time that your program fetches the rows produced by that query, another user's program might execute the following update:

```
EXEC SQL UPDATE stock
    SET unit_price = 1.15 * unit_price
        WHERE manu_code = 'ANZ';
```

In other words, both programs are reading through the same table, one fetching certain rows and the other changing the same rows. The following possibilities are concerned with what happens next:

1. The other program finishes its update before your program fetches its first row.

   Your program shows you only updated rows.

2. Your program fetches every row before the other program has a chance to update it.

   Your program shows you only original rows.

3. After your program fetches some original rows, the other program catches up and goes on to update some rows that your program has yet to read; then it executes the COMMIT WORK statement.

   Your program might return a mixture of original rows and updated rows.

4. Same as number 3, except that after updating the table, the other program issues a ROLLBACK WORK statement.

   Your program can show you a mixture of original rows and updated rows that no longer exist in the database.

The first two possibilities are harmless. In number 1, the update is complete before your query begins. It makes no difference whether the update finished a microsecond ago or a week ago.

In number 2, your query is, in effect, complete before the update begins. The other program might have been working just one row behind yours, or it might not start until tomorrow night; it does not matter.

The last two possibilities, however, can be very important to the design of some applications. In number 3, the query returns a mix of updated and original data. That result can be detrimental in some applications. In others, such as one that is taking an average of all prices, it might not matter at all.

In number 4, it can be disastrous if a program returns some rows of data that, because their transaction was cancelled, can no longer be found in the table.

Another concern arises when your program uses a cursor to update or delete the last-fetched row. Erroneous results occur with the following sequence of events:

- Your program fetches the row.
- Another program updates or deletes the row.
- Your program updates or deletes WHERE CURRENT OF *names*.

To control concurrent events such as these, use the locking and *isolation level* features of the database server.

How Locks Work

Informix database servers support a complex, flexible set of locking features that this section describes. For a summary of locking features, see your *Getting Started* manual.

Kinds of Locks

The following table shows the types of locks that Informix database servers support for different situations.

Lock Type	Use
Shared	A shared lock reserves its object for reading only. It prevents the object from changing while the lock remains. More than one program can place a shared lock on the same object.
Exclusive	An exclusive lock reserves its object for the use of a single program. This lock is used when the program intends to change the object. An exclusive lock cannot be placed where any other kind of lock exists. Once a lock has been placed, no other lock can be placed on the same object.
Promotable	A promotable lock establishes the intent to update. It can only be placed where no other promotable or exclusive lock exists. Promotable locks can be placed on records that already have shared locks. When the program is about to change the locked object, the promotable lock can be promoted to an exclusive lock, but only if no other locks, including shared locks, are on the record at the time the lock would change from promotable to exclusive. If a shared lock was on the record when the promotable lock was set, the shared lock must be dropped before the promotable lock can be promoted to an exclusive lock.

Lock Scope

You can apply locks to entire databases, entire tables, disk pages, single rows, or index-key values. The size of the object that is being locked is referred to as the *scope* of the lock (also called the *lock granularity*). In general, the larger the scope of a lock, the more concurrency is reduced, but the simpler programming becomes.

Database Locks

You can lock an entire database. The act of opening a database places a shared lock on the name of the database. A database is opened with the CONNECT, DATABASE, or CREATE DATABASE statements. As long as a program has a database open, the shared lock on the name prevents any other program from dropping the database or putting an exclusive lock on it.

You can lock an entire database exclusively with the following statement:

```
DATABASE database name EXCLUSIVE
```

This statement succeeds if no other program has opened that database. Once the lock is placed, no other program can open the database, even for reading, because its attempt to place a shared lock on the database name fails.

A database lock is released only when the database closes. That action can be performed explicitly with the DISCONNECT or CLOSE DATABASE statements or implicitly by executing another DATABASE statement.

Because locking a database reduces concurrency in that database to zero, it makes programming very simple; concurrent effects cannot happen. However, you should lock a database only when no other programs need access. Database locking is often used before applying massive changes to data during off-peak hours.

Table Locks

You can lock entire tables. In some cases, this action is performed automatically. The database server always locks an entire table while it performs any of the following statements:

- ALTER FRAGMENT
- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- RENAME COLUMN
- RENAME TABLE

AD/XP

Dynamic Server with AD and XP Options does not support the ALTER INDEX statement. ♦

Completion of the statement (or end of the transaction) releases the lock. An entire table can also be locked automatically during certain queries.

You can use the LOCK TABLE statement to lock an entire table explicitly. This statement allows you to place either a shared lock or an exclusive lock on an entire table.

A shared table lock prevents any concurrent updating of that table while your program is reading from it. The database server achieves the same degree of protection by setting the isolation level, as described in the next section, which allows greater concurrency than using a shared table lock. However, all Informix database servers support the LOCK TABLE statement.

An exclusive table lock prevents any concurrent use of the table and, therefore, can have a serious effect on performance if many other programs are contending for the use of the table. Like an exclusive database lock, an exclusive table lock is often used when massive updates are applied during off-peak hours. For example, some applications do not update tables during the hours of peak use. Instead, they write updates to an *update journal*. During off-peak hours, that journal is read, and all updates are applied in a batch.

AD/XP

TABLE Lock Mode for Informix Dynamic Server with AD and XP Options

With Dynamic Server with AD and XP Options, you can lock a table with either the LOCK TABLE statement or the TABLE lock mode of the LOCK MODE clause. All transactions that access a table whose lock mode is set to TABLE acquire a table lock for that table, if the isolation level for the transaction requires the transaction to acquire any locks at all.

Use the ALTER TABLE statement to switch a table from one lock mode to any other lock mode (TABLE, PAGE, or ROW).

Whether you specify the TABLE lock mode for the LOCK MODE clause of a CREATE TABLE or ALTER TABLE statement, or use a LOCK TABLE statement to acquire a table lock, the effect is the same.

The TABLE lock mode is particularly useful in a data-warehousing environment where query efficiency increases because, instead of acquiring (or trying to acquire, depending on the isolation level) page- or row-level locks, the transaction acquires table locks. This can significantly reduce the number of lock requests. The disadvantage of table locks is that they radically reduce update concurrency, but in a data warehousing environment this is generally not a problem.

Page, Row, and Key Locks

One row of a table is the smallest object that can be locked. A program can lock one row or a selection of rows while other programs continue to work on other rows of the same table.

The database server stores data in units called *disk pages*. A disk page contains one or more rows. In some cases, it is better to lock a disk page than to lock individual rows on it. Disk-storage methods for your database server are described in your *Administrator's Guide*. Tips for optimizing tables on disk storage can be found in your *Performance Guide*.

You choose between locking by rows or locking by pages when you create the table. The database server supports a LOCK MODE clause to specify either page or row locking. You can specify lock mode in the CREATE TABLE statement and later change it with the ALTER TABLE statement.

You use page and row locking identically. Whenever the database server needs to lock a row, it locks either the row itself or the page it is on, depending on the lock mode established for the table.

In certain cases, the database server has to lock a row that does not exist. In effect, it locks the place in the table where the row would be if it did exist. The database server does this by placing a lock on an index-key value. Key locks are used identically to row locks. When the table uses row locking, key locks are implemented as locks on imaginary rows. When the table uses page locking, a key lock is placed on the index page that contains the key or that would contain the key if it existed.

Duration of a Lock

The program controls the duration of a database lock. A database lock is released when the database closes.

Depending on whether the database uses transactions, table lock durations will vary. If the database does not use transactions (that is, if no transaction log exists and you do not use COMMIT WORK statement), a table lock remains until it is removed by the execution of the UNLOCK TABLE statement.

The duration of table, row, and index locks depends on what SQL statements you use and on whether transactions are in use.

When you use transactions, the end of a transaction releases all table, row, page, and index locks. When a transaction ends, *all locks are released*.

Locks While Modifying

When the database server fetches a row through an update cursor, it places a promotable lock on the fetched row. If this action succeeds, the database server knows that no other program can alter that row. Because a promotable lock is not exclusive, other programs can continue to read the row. This helps performance because the program that fetched the row can take some time before it issues the UPDATE or DELETE statement, or it can simply fetch the next row.

When it is time to modify a row, the database server obtains an exclusive lock on the row. If it already had a promotable lock, it changes that lock to exclusive status.

The duration of an exclusive row lock depends on whether transactions are in use. If they are not in use, the lock is released as soon as the modified row is written to disk. When transactions are in use, all such locks are held until the end of the transaction. This action prevents other programs from using rows that might be rolled back to their original state.

When transactions are in use, a key lock is used whenever a row is deleted. Using a key lock prevents the following error from occurring:

- Program A deletes a row.
- Program B inserts a row that has the same key.
- Program A rolls back its transaction, forcing the database server to restore its deleted row.

  What is to be done with the row inserted by Program B?

By locking the index, the database server prevents a second program from inserting a row until the first program commits its transaction.

The locks placed while the database reads various rows are controlled by the current isolation level, which is discussed in the next section.

Setting the Isolation Level

The *isolation level* is the degree to which your program is isolated from the concurrent actions of other programs. The database server offers a choice of isolation levels. It implements them by setting different rules for how a program uses locks when it is reading. (This description does not apply to reads performed on update cursors.)

To set the isolation level, use either the SET ISOLATION or SET TRANSACTION statement. The SET TRANSACTION statement also lets you set access modes. For more information about access modes, see "Controlling Data Modification with Access Modes" on page 7-18.

Comparing SET TRANSACTION with SET ISOLATION

The SET TRANSACTION statement complies with ANSI SQL-92. This statement is similar to the Informix SET ISOLATION statement; however, the SET ISOLATION statement is not ANSI compliant and does not provide access modes.

The isolation levels that you can set with the SET TRANSACTION statement are comparable to the isolation levels that you can set with the SET ISOLATION statement, as the following table shows.

SET TRANSACTION	Correlates to	SET ISOLATION
Read Uncommitted		Dirty Read
Read Committed		Committed Read
Not Supported		Cursor Stability
(ANSI) Repeatable Read Serializable		(Informix) Repeatable Read (Informix) Repeatable Read

The major difference between the SET TRANSACTION and SET ISOLATION statements is the behavior of the isolation levels within transactions. The SET TRANSACTION statement can be issued only once for a transaction. Any cursors opened during that transaction are guaranteed to have that isolation level (or access mode if you are defining an access mode). With the SET ISOLATION statement, after a transaction is started, you can change the isolation level more than once within the transaction. The following examples show both the SET ISOLATION and SET TRANSACTION statements:

SET ISOLATION

```
EXEC SQL BEGIN WORK;
EXEC SQL SET ISOLATION TO DIRTY READ;
EXEC SQL SELECT ... ;
EXEC SQL SET ISOLATION TO REPEATABLE READ;
EXEC SQL INSERT ... ;
EXEC SQL COMMIT WORK;
    -- Executes without error
```

SET TRANSACTION

```
EXEC SQL BEGIN WORK;
EXEC SQL SET TRANSACTION ISOLATION LEVEL TO SERIALIZABLE;
EXEC SQL SELECT ... ;
EXEC SQL SET TRANSACTION ISOLATION LEVEL TO READ COMMITTED;
Error 876: Cannot issue SET TRANSACTION more than once in an
active transaction.
```

ANSI Read Uncommitted and Informix Dirty Read Isolation

The simplest isolation level, ANSI Read Uncommitted and Informix Dirty Read, amounts to virtually no isolation. When a program fetches a row, it places no locks, and it respects none; it simply copies rows from the database without regard for what other programs are doing.

A program always receives complete rows of data; even under ANSI Read Uncommitted or Informix Dirty Read isolation, a program never sees a row in which some columns are updated and some are not. However, a program that uses ANSI Read Uncommitted or Informix Dirty Read isolation sometimes reads updated rows before the updating program ends its transaction. If the updating program later rolls back its transaction, the reading program processed data that never really existed (number 4 in the list of concurrency issues on page 7-5).

ANSI Read Uncommitted or Informix Dirty Read is the most efficient isolation level. The reading program never waits and never makes another program wait. It is the preferred level in any of the following cases:

- All tables are static; that is, concurrent programs only read and never modify data.
- The table is held in an exclusive lock.
- Only one program is using the table.

ANSI Read Committed and Informix Committed Read Isolation

When a program requests the ANSI Read Committed or Informix Committed Read isolation level, the database server guarantees that it never returns a row that is not committed to the database. This action prevents reading data that is not committed and that is subsequently rolled back.

ANSI Read Committed or Informix Committed Read is implemented simply. Before it fetches a row, the database server tests to determine whether an updating process placed a lock on the row; if not, it returns the row. Because rows that are updated but not committed have locks on them, this test ensures that the program does not read uncommitted data.

ANSI Read Committed or Informix Committed Read does not actually place a lock on the fetched row, so this isolation level is almost as efficient as ANSI Read Uncommitted or Informix Dirty Read. This isolation level is appropriate to use when each row of data is processed as an independent unit, without reference to other rows in the same or other tables.

Informix Cursor Stability Isolation

The next level, Cursor Stability, is available only with the Informix SQL statement SET ISOLATION.

IDS

When Cursor Stability is in effect, Dynamic Server places a lock on the latest row fetched. It places a shared lock for an ordinary cursor or a promotable lock for an update cursor. Only one row is locked at a time; that is, each time a row is fetched, the lock on the previous row is released (unless that row is updated, in which case the lock holds until the end of the transaction). ♦

AD/XP

When Cursor Stability is in effect, Dynamic Server with AD and XP Options places a lock on one or more rows. It places a shared lock for an ordinary cursor or a promotable lock for an update cursor.You use the ISOLATION_LOCKS configuration parameter to specify the maximum number of rows to be locked at any given time on any given scan. The database server includes the user's current row in the set of rows currently locked. As the next row is read from the cursor, the previous row might or might not be released. The user does not have control over which rows are locked or when those rows are released. The database server guarantees only that a maximum of *n* rows are locked at any given time for any given cursor and that the current row is in the set of rows currently locked. (The default value is one row.) For more information about the ISOLATION_LOCKS parameter, see your *Performance Guide* and *Administrator's Guide*. ♦

Cursor Stability ensures that a row does not change while the program examines it. Such row stability is important when the program updates some other table based on the data it reads from the row. Because of Cursor Stability, the program is assured that the update is based on current information. It prevents the use of *stale data*.

The following example illustrates effective use of Cursor Stability isolation. In terms of the demonstration database, Program A wants to insert a new stock item for manufacturer Hero (HRO). Concurrently, Program B wants to delete manufacturer HRO and all stock associated with it. The following sequence of events can occur:

1. Program A, operating under Cursor Stability, fetches the HRO row from the **manufact** table to learn the manufacturer code: This action places a shared lock on the row.
2. Program B issues a DELETE statement for that row. Because of the lock, the database server makes the program wait.
3. Program A inserts a new row in the **stock** table using the manufacturer code it obtained from the **manufact** table.
4. Program A closes its cursor on the **manufact** table or reads a different row of it, releasing its lock.
5. Program B, released from its wait, completes the deletion of the row and goes on to delete the rows of **stock** that use manufacturer code HRO, including the row just inserted by Program A.

If Program A used a lesser level of isolation, the following sequence could occur:

1. Program A reads the HRO row of the **manufact** table to learn the manufacturer code. No lock is placed.
2. Program B issues a DELETE statement for that row. It succeeds.
3. Program B deletes all rows of **stock** that use manufacturer code HRO.
4. Program B ends.
5. Program A, not aware that its copy of the HRO row is now invalid, inserts a new row of **stock** using the manufacturer code HRO.
6. Program A ends.

At the end, a row occurs in **stock** that has no matching manufacturer code in **manufact**. Furthermore, Program B apparently has a bug; it did not delete the rows that it was supposed to delete. Use of the Cursor Stability isolation level prevents these effects.

The preceding scenario could be rearranged to fail even with Cursor Stability. All that is required is for Program B to operate on tables in the reverse sequence to Program A. If Program B deletes from **stock** before it removes the row of **manufact**, no degree of isolation can prevent an error. Whenever this kind of error is possible, all programs that are involved must use the same sequence of access.

Because Cursor Stability locks only one row (Dynamic Server) or a specified number of rows (Dynamic Server with AD and XP Options) at a time, it restricts concurrency less than a table lock or database lock.

ANSI Serializable, ANSI Repeatable Read, and Informix Repeatable Read Isolation

The definitions for ANSI Serializable, ANSI Repeatable Read, and Informix Repeatable Read isolation levels are all the same.

The Repeatable Read isolation level asks the database server to put a lock on every row the program examines and fetches. The locks that are placed are shareable for an ordinary cursor and promotable for an update cursor. The locks are placed individually as each row is examined. They are not released until the cursor closes or a transaction ends.

Repeatable Read allows a program that uses a scroll cursor to read selected rows more than once and to be sure that they are not modified or deleted between readings. (Scroll cursors are described in Chapter 5, "Programming with SQL.") No lower isolation level guarantees that rows still exist and are unchanged the second time they are read.

Repeatable Read isolation places the largest number of locks and holds them the longest. Therefore, it is the level that reduces concurrency the most. If your program uses this level of isolation, think carefully about how many locks it places, how long they are held, and what the effect can be on other programs.

In addition to the effect on concurrency, the large number of locks can be a problem. The database server records the number of locks by each program in a lock table. If the maximum number of locks is exceeded, the lock table fills up, and the database server cannot place a lock. An error code is returned. The person who administers an Informix database server system can monitor the lock table and tell you when it is heavily used.

The isolation level in an ANSI-compliant database is automatically set to Serializable. The isolation level of Serializable is required to ensure that operations behave according to the ANSI standard for SQL.

Controlling Data Modification with Access Modes

Informix database servers support access modes. Access modes affect read and write concurrency for rows within transactions and are set with the SET TRANSACTION statement. You can use access modes to control data modification among shared files.

Transactions are read-write by default. If you specify that a transaction is read-only, that transaction cannot perform the following tasks:

- Insert, delete, or update table rows
- Create, alter, or drop any database object such as a schema, table, temporary table, index, or stored procedure
- Grant or revoke privileges
- Update statistics
- Rename columns or tables

Read-only access mode prohibits updates.

You can execute stored procedures in a read-only transaction as long as the procedure does not try to perform any restricted statements.

Setting the Lock Mode

The lock mode determines what happens when your program encounters locked data. One of the following situations occurs when a program attempts to fetch or modify a locked row:

- The database server immediately returns an error code in SQLCODE or SQLSTATE to the program.
- The database server suspends the program until the program that placed the lock removes the lock.
- The database server suspends the program for a time and then, if the lock is not removed, the database server sends an error-return code to the program.

You choose among these results with the SET LOCK MODE statement.

Waiting for Locks

If you prefer to wait (this choice is best for many applications), execute the following statement:

```
SET LOCK MODE TO WAIT
```

When this lock mode is set, your program usually ignores the existence of other concurrent programs. When your program needs to access a row that another program has locked, it waits until the lock is removed, then proceeds. The delays are usually imperceptible.

Not Waiting for Locks

The disadvantage of waiting for locks is that the wait might become long (although properly designed applications should hold their locks briefly). When the possibility of a long delay is not acceptable, a program can execute the following statement:

```
SET LOCK MODE TO NOT WAIT
```

When the program requests a locked row, it immediately receives an error code (for example, error -107 `Record is locked`), and the current SQL statement terminates. The program must roll back its current transaction and try again.

The initial setting is *not waiting* when a program starts up. If you are using SQL interactively and see an error related to locking, set the lock mode to wait. If you are writing a program, consider making that one of the first embedded SQL statements that the program executes.

Waiting a Limited Time

You can ask the database server to set an upper limit on a wait. You can issue the following statement:

```
SET LOCK MODE TO WAIT 17
```

This statement places an upper limit of 17 seconds on the length of any wait. If a lock is not removed in that time, the error code is returned.

Handling a Deadlock

A *deadlock* is a situation in which a pair of programs block the progress of each other. Each program has a lock on some object that the other program wants to access. A deadlock arises only when all programs concerned set their lock modes to wait for locks.

An Informix database server detects deadlocks immediately when they involve only data at a single network server. It prevents the deadlock from occurring by returning an error code (error -143 `ISAM error: deadlock detected`) to the second program to request a lock. The error code is the one the program receives if it sets its lock mode to not wait for locks. If your program receives an error code related to locks even after it sets lock mode to wait, you know the cause is an impending deadlock.

Handling External Deadlock

A deadlock can also occur between programs on different database servers. In this case, the database server cannot instantly detect the deadlock. (Perfect deadlock detection requires excessive communications traffic among all database servers in a network.) Instead, each database server sets an upper limit on the amount of time that a program can wait to obtain a lock on data at a different database server. If the time expires, the database server assumes that a deadlock was the cause and returns a lock-related error code.

In other words, when external databases are involved, every program runs with a maximum lock-waiting time. The database administrator can set or modify the maximum for the database server.

Simple Concurrency

If you are not sure which choice to make concerning locking and concurrency, and if your application is straightforward, have your program execute the following statements when it starts up (immediately after the first DATABASE statement):

```
SET LOCK MODE TO WAIT
SET ISOLATION TO REPEATABLE READ
```

Ignore the return codes from both statements. Proceed as if no other programs exist. If no performance problems arise, you do not need to read this section again.

Hold Cursors

When transaction logging is used, all database servers (except Dynamic Server with AD and XP Options) guarantee that anything done within a transaction can be rolled back at the end of it. To handle transactions reliably, the database server normally applies the following rules:

- When a transaction ends all cursors are closed.
- When a transaction ends all locks are released.

AD/XP

With Dynamic Server with AD and XP Options, locks might not be released at the end of a transaction. To demonstrate how to acquire a table lock, suppose the database server acquires a lock on all coservers that store a part of the table. If a transaction first acquires a SHARED mode table lock and tries to upgrade to EXCLUSIVE mode table lock, locks might not be released at the end of the transaction. This can happen if the transaction performs a SELECT and then performs an INSERT on a table with lock mode TABLE. In this case, the upgrade might succeed on some coservers and fail on other coservers. No attempt is made to roll back the successful upgrades, which means that the transaction might end with EXCLUSIVE locks on the table for some coservers. ♦

The rules that are used to handle transactions reliably are normal with most database systems that support transactions, and they do not cause any trouble for most applications. However, circumstances exist in which using standard transactions with cursors is not possible. For example, the following code works fine without transactions. However, when transactions are added, closing the cursor conflicts with using two cursors simultaneously.

```
EXEC SQL DECLARE master CURSOR FOR . . .
EXEC SQL DECLARE detail CURSOR FOR . . . FOR UPDATE
EXEC SQL OPEN master;
while(SQLCODE == 0)
{
    EXEC SQL FETCH master INTO . . .
    if(SQLCODE == 0)
    {
        EXEC SQL BEGIN WORK;
        EXEC SQL OPEN detail USING . . .
        EXEC SQL FETCH detail . . .
        EXEC SQL UPDATE . . . WHERE CURRENT OF detail
        EXEC SQL COMMIT WORK;
    }
}
EXEC SQL CLOSE master;
```

In this design, one cursor is used to scan a table. Selected records are used as the basis for updating a different table. The problem is that when each update is treated as a separate transaction (as the pseudocode in the previous example shows), the COMMIT WORK statement following the UPDATE closes all cursors, including the master cursor.

The simplest alternative is to move the COMMIT WORK and BEGIN WORK statements to be the last and first statements, respectively, so that the entire scan over the master table is one large transaction. Treating the scan of the master table as one large transaction is sometimes possible, but it can become impractical if many rows need to be updated. The number of locks can be too large, and they are held for the duration of the program.

A solution that Informix database servers support is to add the keywords WITH HOLD to the declaration of the master cursor. Such a cursor is referred to as a *hold cursor* and is not closed at the end of a transaction. The database server still closes all other cursors, and it still releases all locks, but the hold cursor remains open until it is explicitly closed.

Before you attempt to use a hold cursor, you must be sure that you understand the locking mechanism described here, and you must also understand the programs that are running concurrently. Whenever COMMIT WORK is executed, all locks are released, including any locks placed on rows fetched through the hold cursor.

The removal of locks has little importance if the cursor is used as intended, for a single forward scan over a table. However, you can specify WITH HOLD for any cursor, including update cursors and scroll cursors. Before you do this, you must understand the implications of the fact that all locks (including locks on entire tables) are released at the end of a transaction.

Summary

Whenever multiple programs have access to a database concurrently (and when at least one of them can modify data), all programs must allow for the possibility that another program can change the data even as they read it. The database server provides a mechanism of locks and isolation levels that usually allow programs to run as if they were alone with the data.

Section II

Using Advanced SQL

Chapter

Creating and Using Stored Procedures

8

Introduction to Stored Procedures and SPL 8-3
What You Can Do with Stored Procedures 8-4
Relationship Between SQL and a Stored Procedure 8-4

Stored-Procedure Behavior for Dynamic Server
with AD and XP Options 8-5

Creating and Using Stored Procedures 8-6
Creating a Procedure 8-6
Creating a Procedure in a Program 8-7
Commenting and Documenting a Procedure 8-8
Diagnosing Compile-Time Errors 8-8
Using DB-Access to Find Syntax Errors in a Procedure 8-8
Using an SQL API to Find Syntax Errors in a Procedure . . . 8-9
Looking at Compile-Time Warnings 8-10
Location of the Listing File 8-10
Viewing the Listing File 8-11
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
Executing a Stored Procedure Dynamically 8-14
Debugging a Procedure 8-14
Re-creating a Procedure 8-16

Privileges on Stored Procedures 8-17
Privileges Necessary at Creation 8-18
Privileges Necessary at Execution 8-18
Owner-Privileged Procedures 8-18
DBA-Privileged Procedures 8-19
Privileges and Nested Procedures 8-19
Revoking Privileges 8-20

Variables and Expressions 8-20
SPL Variables 8-20
Local Variables. 8-20
Global Variables 8-21
Format of Variables 8-21
Defining Variables 8-21
Data Types for Variables 8-22
Using Subscripts with Variables 8-22
Scope of Variables. 8-22
Variable and Keyword Ambiguity 8-23
SPL Expressions 8-25
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 a Procedure . . 8-28
Calling a Procedure Recursively. 8-29

Passing Information to and from a Procedure 8-29
Returning Results 8-30
Specifying Return Data Types 8-30
Returning the Value 8-30
Returning More Than One Set of Values from a Procedure. . . 8-31

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

Summary 8-36

You can use SQL and some additional statements that belong to the Stored Procedure Language (SPL) to write procedures and store the procedures in the database. These stored procedures are effective tools for controlling SQL activity. This chapter shows how to write stored procedures. To help you learn how to write them, examples of working stored procedures are provided.

The syntax for each SPL statement is described in the *Informix Guide to SQL: Syntax*. The syntax includes a description of the SPL statement and examples that show how to use the statement.

Introduction to Stored Procedures and SPL

In SQL, a stored procedure is a user-defined function. Anyone who has the 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 that you can perform in SQL as well as to expand what you can accomplish with SQL alone.

You use SQL and SPL statements to write a stored procedure. SPL statements can be used only inside the CREATE PROCEDURE and CREATE PROCEDURE FROM statements. The CREATE PROCEDURE statement is available with DB-Access and Relational Object Manager. Both the CREATE PROCEDURE and CREATE PROCEDURE FROM statements are available with SQL APIs such as INFORMIX-ESQL/C.

What You Can Do with Stored Procedures

You can accomplish a wide range of objectives with stored procedures, including improving database performance, simplifying writing applications, and limiting or monitoring access to data.

Because a stored procedure is stored in an executable format, you can use it to execute frequently repeated tasks to improve performance. When you execute a stored procedure rather than straight SQL code you can bypass repeated parsing, validity checking, and query optimization.

Because a stored procedure is an object in the database, it is available to every application that runs on the database. Several applications can use the same stored procedure, so development time for applications is reduced.

You can write a stored procedure to be run with the DBA privilege by a user who does not have the DBA privilege. This feature allows you to limit and control access to data in the database. Alternatively, a stored procedure can monitor the users who access certain tables or data. For more information about how to use stored procedures to control access to data, see the *Informix Guide to Database Design and Implementation*.

Relationship Between SQL and a Stored Procedure

You can call a procedure in data-manipulation SQL statements and issue SQL statements within a procedure. For a complete list of data-manipulation SQL statements, see the *Informix Guide to SQL: Syntax*.

You use a stored procedure in a data-manipulation SQL statement to supply values to that statement. For example, you can use a procedure to perform the following actions:

- Supply values to be inserted into a table
- Supply a value that makes up part of a condition clause in a SELECT, DELETE, or UPDATE statement

These actions are two possible uses of a procedure in a data-manipulation statement, but others exist. In fact, any expression in a data-manipulation SQL statement can consist of a procedure call.

You can also issue SQL statements in a stored procedure to hide those SQL statements from 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 so that they can execute it.

AD/XP

Stored-Procedure Behavior for Dynamic Server with AD and XP Options

For stored procedures in Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options, the following features behave differently than they do in other Informix database servers:

- SYSPROCPLAN system catalog table

  All Informix database servers modify the SYSPROCPLAN system catalog table whenever a stored procedure is created. For all database servers except Dynamic Server with AD and XP Options, the SYSPROCPLAN system catalog table is also modified during execution of a stored procedure, if the stored procedure generates any new query-execution plans during execution. However, Dynamic Server with AD and XP Options does not modify the SYSPROCPLAN table when execution of a stored procedure results in new query-execution plans. For example, if plans are deleted from the SYSPROCPLAN system catalog table, and the procedure is executed from any coserver, the plans are not restored in SYSPROCPLAN. However, an UPDATE STATISTICS FOR PROCEDURE statement that is executed from any coserver updates the plans in SYSPROCPLAN.
- Procedure calls

  A procedure call can be made only to procedures that are in the current database and the current database server.

Creating and Using Stored Procedures

To write a stored procedure, put the SQL statements that you want to run as part of the procedure in the statement block in a CREATE PROCEDURE statement. You can use SPL statements to control the flow of the operation in the procedure. SPL statements include IF, FOR, and others, and they are described in the *Informix Guide to SQL: Syntax*. The CREATE PROCEDURE and CREATE PROCEDURE FROM statements are also described in the *Informix Guide to SQL: Syntax*.

Creating a Procedure

To create a stored procedure with DB-Access or Relational Object Manager, issue the CREATE PROCEDURE statement, including all the statements that are part of the procedure in the statement block. For example, to create a procedure that reads a customer address, use a statement such as the following one:

```
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 'pathname' -- modify this pathname according
-- to the conventions that your operating system requires

-- compile-time warnings go here
; -- end of the procedure read_address
```

Creating a Procedure in a Program

To use an SQL API to create a stored procedure, 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 the previous example 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';
```

The following example shows how the previous SQL statement looks in an ESQL/C program:

```
/* This program creates whatever procedure is in *
 * the file 'read_add_source'.
 */
#include <stdio.h>
EXEC SQL include sqlca;
EXEC SQL include sqlda;
EXEC SQL include datetime;
/* Program to create a procedure from the pwd */

main()
{
EXEC SQL database play;
EXEC SQL create procedure from 'read_add_source';
}
```

Commenting and Documenting a Procedure

Observe that the **read_address** procedure in the previous example includes comments and a DOCUMENT clause. The programmer incorporates the comments into the text of the procedure. Any characters that follow a double hyphen (--) are considered to be a comment. You can use the double hyphen anywhere in a line.

The text in the DOCUMENT clause should give a summary of the procedure. To extract this text, query the **sysprocbody** system catalog table. For information about how to read the DOCUMENT clause, see "Looking at the Procedure Documentation" on page 8-11.

Diagnosing Compile-Time Errors

When you issue a CREATE PROCEDURE or CREATE PROCEDURE FROM statement, the statement fails if a syntax error occurs in the body of the procedure. The database server stops processing the text of the procedure and returns the location of the error.

Using DB-Access to Find Syntax Errors in a Procedure

If a procedure that you create with DB-Access or Relational Object Manager has a syntax error, when you choose the Modify option of the SQL menu, the cursor sits on the line that contains the offending syntax.

Using an SQL API to Find Syntax Errors in a Procedure

If a procedure that you created with an SQL API has a syntax error, the CREATE PROCEDURE statement fails and sets SQLCA and SQLSTATE values. 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 following list shows the particular fields of the SQLCA for ESQL/C:

- sqlca.sqlcode (SQLCODE)
- sqlca.sqlerrd[4]

In case of syntax error, the database server sets SQLSTATE to 42000.

The following example shows how to trap for a syntax error when you are creating a procedure. It also shows how to display a message and character position where the error occurred.

```
#include <stdio.h>
EXEC SQL include sqlca;
EXEC SQL include sqlda;
EXEC SQL include datetime;
/* Program to create a procedure from procfile in pwd */

main()
{
long char_num;

EXEC SQL database play;
EXEC SQL 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 the previous example, if the CREATE PROCEDURE FROM statement fails, the program displays a message in addition to the character position at which the syntax error occurred.

Looking at Compile-Time Warnings

If the database server detects a potential problem, but the procedure is syntactically correct, the database server generates a warning and places it in a listing file. You can examine this file to check for potential problems before you execute the procedure.

To obtain the listing of compile-time warnings for your procedure, use the WITH LISTING IN clause in your CREATE PROCEDURE statement, as the following example shows:

```
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 'pathname' -- modify this pathname according to the conventions
                    -- that your operating system requires


--compile-time warnings go here
; -- end of the procedure read_address
```

Location of the Listing File

If you are working on a network, the listing file is created on the computer where the database resides. If you provide an absolute pathname and filename for the file, the file is created where you specify.

UNIX

If you provide a relative pathname for the listing file, the file is created in your home directory on the computer where the database resides. (If you do not have a home directory, the file is created in the **root** directory.) ♦

WIN NT

If you provide a relative pathname for the listing file, the default directory is your current working directory if the database is on the local computer. Otherwise the default directory is **%INFORMIXDIR%\bin**. ♦

Viewing the Listing File

After you create the procedure, you can view the file that is specified in the WITH LISTING IN clause to see the warnings that 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.sysprocedures
        WHERE informix.sysprocedures.procname = 'read_address')
```

Looking at the Procedure Documentation

If you want to view only the documenting text of the procedure, use the following SELECT statement to read the documentation string. The documentation lines found in the following example 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.sysprocedures.procname = 'read_address')
```

Executing a Procedure

You can execute a procedure in several ways. You can use either the SQL statement EXECUTE PROCEDURE or the LET or CALL SPL statement. In addition, you can execute procedures dynamically, as described in "Executing a Stored Procedure Dynamically" on page 8-14.

The **read_address** procedure returns the full name and address of a customer. To run **read_address** on a customer called Putnum with EXECUTE PROCEDURE, enter the following statement:

```
EXECUTE PROCEDURE read_address ('Putnum');
```

The **read_address** procedure returns values; therefore, if you are executing a procedure from an SQL API or 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 that the following example shows:

```
#include <stdio.h>
EXEC SQL include sqlca;
EXEC SQL include sqlda;
EXEC SQL include datetime;
/* Program to execute a procedure in the database named 'play'
*/

main()
{
EXEC SQL BEGIN DECLARE SECTION;
    char lname[16], fname[16], address[21];
    char city[16], state[3], zip[6];
EXEC SQL END DECLARE SECTION;
EXEC SQL connect to 'play';
EXEC SQL 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 in another procedure, you can use the SPL statements CALL or LET to run the procedure. To use the CALL statement with the **read_address** procedure, you can use the code in the following example:

```
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;
```

The following example shows how to use the LET statement to assign values to procedural variables through a procedure call:

```
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;
```

AD/XP

With Dynamic Server with AD and XP Options, you can make a procedure call only to procedures that are in the current database and the current database server. ♦

Executing a Stored Procedure Dynamically

You can prepare an EXECUTE PROCEDURE statement with the ALLOCATE DESCRIPTOR and GET DESCRIPTOR statements in an ESQL/C program. You can pass parameters to the stored procedure in the same manner as the SELECT statement and can pass them at runtime or compile time. For an example of how to execute a stored procedure dynamically, see the *INFORMIX-ESQL/C Programmer's Manual*. For information about dynamic SQL and how to use a prepared SELECT statement, see Chapter 5, "Programming with SQL."

Debugging a Procedure

Once you successfully create and run a procedure, you can encounter logic errors. If the procedure contains logic errors, use the TRACE statement to help find them. You can trace the values of the following procedural entities:

- Variables
- Procedure arguments
- Return values
- SQL error codes
- ISAM error codes

To generate a list 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.

The following methods specify the form of TRACE output.

Statement	Action
TRACE ON	Traces all statements except SQL statements. Prints the contents of variables are printed before they are used. Traces procedure calls and returned values.
TRACE PROCEDURE	Traces only the procedure calls and returned values.
TRACE *expression*	Prints a literal or an expression. If necessary, the value of the expression is calculated before it is sent to the file.

The following example shows how you can use the TRACE statement with a version of the **read_address** procedure. This example shows several SPL statements that have not been discussed, but the entire example demonstrates how the TRACE statement can help you monitor execution of the 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 with any text editor.

The following table contains some of the output that the procedure in the previous example generates. Next to each traced statement is an explanation of its contents.

Statement	Action
`TRACE ON`	Echoes TRACE ON statement.
`TRACE Foreach starts`	Traces expression, in this case, the literal string `Foreach starts`.
`start select cursor`	Provides notification that a cursor is opened to handle a FOREACH loop.
`select cursor iteration`	Provides notification of the start of each iteration of the select cursor.
`expression: (+lcount, 1)`	Evaluates the encountered expression, `(lcount+1)`, to 2.
`let lcount = 2`	Echoes each LET statement 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

A stored procedure resides in the database in which it is created. As with other database objects, you need appropriate privileges to create a stored procedure. In addition, you need appropriate privileges to execute a stored procedure.

Two types of stored procedures exist in a database: DBA-privileged and owner-privileged. When you create the procedure, you specify which type it is. You need different privileges to create and execute these two types of procedures. The differences between DBA-privileged and owner-privileged procedures are summarized in the following table and described in the sections that follow.

	DBA-Privileged Procedure	**Owner-Privileged Procedure**
Can be created by:	Any user with the DBA privilege	Any user with at least the Resource privilege
Users who have the Execute privilege by default:	Any user with the DBA privilege	Not ANSI compliant. Public (any user with Connect database privilege) ANSI compliant. The procedure owner and any user with the DBA privilege
Privileges the procedure owner or WITH clause must grant to another user to enable that user to run a procedure:	Execute privilege	Execute privilege and privileges on underlying objects If owner has privileges on underlying objects with the GRANT WITH option, only the Execute privilege is required.

Privileges Necessary at Creation

Only users who have the DBA privilege can create a DBA-privileged procedure. To create an owner-privileged procedure, you need to have at least the Resource privilege. For more information about how to grant and limit access to your database, see the *Informix Guide to Database Design and Implementation*.

Privileges Necessary at Execution

To run a procedure, you always need the Execute privilege for that procedure or DBA database privileges. The database server implicitly grants certain privileges to users, depending on whether the procedure is a DBA-mode procedure and if the database is ANSI compliant.

If the procedure is owner privileged, the database server grants the Execute privilege to Public. If the database is ANSI compliant, the database server grants only the Execute privilege to the owner and users with DBA status.

If the procedure is DBA privileged, the database server grants the Execute privilege to all users who have the DBA privilege.

Owner-Privileged Procedures

When you execute an owner-privileged procedure, the database server checks the existence of any referenced objects. In addition, the database server verifies that you have the necessary privileges on the referenced objects.

If you execute a procedure that references only objects that you own, no privilege conflicts occurs. If you do not own the referenced objects, and you execute a procedure that contains SELECT statements, you risk generating a conflict.

If the owner has the necessary privileges with the WITH GRANT option, those privileges are automatically conferred to you when the owner issues a GRANT EXECUTE statement.

The user who runs the procedure does not own the unqualified objects created in the course of executing the procedure. The owner of the procedure owns the unqualified objects. The following example shows lines in an owner-privileged stored procedure that 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**. The table **gargantuan** is an unqualified name; therefore, **tony** owns the table **gargantuan**. The table **tiny** is qualified by the owner **libby**, so **libby** owns the table **tiny**.

```
CREATE PROCEDURE tryit()
    .
    .
    .
    CREATE TABLE gargantuan (col1 INT, col2 INT, col3 INT);
    CREATE TABLE libby.tiny (col1 INT, col2 INT, col3 INT);

END PROCEDURE;
```

DBA-Privileged Procedures

When you execute a DBA-privileged procedure, you assume the privileges of a DBA for the duration of the procedure. A DBA-privileged procedure acts as if the user who runs the procedure 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 who runs the procedure, unless the data definition statement in the procedure explicitly names the owner to be someone else.

Privileges and Nested Procedures

DBA-privileged status is not inherited by a called procedure. For example, 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 the Execute privilege from a user. If a user loses the Execute privilege on a procedure, the Execute privilege is also revoked from all users who were granted the Execute privilege by that user.

Variables and Expressions

This section discusses how to define and use variables in SPL. This section also describes the differences between SPL and SQL expressions.

SPL Variables

You can use a variable in a stored procedure in several ways. You can use a variable 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.

You can define a variable to be either local or global. A variable is local by default. The following sections describe the differences between the two types of variables.

Local Variables

A local variable is available only in the procedure which defines it. Local variables do not allow a default value to be assigned at compile time.

Global Variables

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 SQL API or in a DB-Access or Relational Object Manager 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.

A global variable that you define in a stored procedure is accessible from all other procedures executed in the same session. When multiple stored procedures modify a global variable, the database server guarantees that only one stored procedure modifies the variable at any given instant.

IDS

Dynamic Server guarantees sequential consistency for global variables. ♦

AD/XP

Dynamic Server with AD and XP Options does not guarantee sequential consistency, and you should not make any assumptions about the order in which the stored procedures are executed. ♦

Format of Variables

A variable follows the rules of an SQL identifier. For information about SQL identifiers, see the *Informix Guide to SQL: Syntax*. Once you define a variable, you can use it anywhere in the procedure as appropriate.

If you are using an SQL API, you do not have to set off the variable with a special symbol (unlike host variables in an SQL API).

Defining Variables

To define variables, use 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 can be null, to a variable before you can use it. For complete information on the DEFINE statement, see the *Informix Guide to SQL: Syntax*.

Data Types for Variables

You can define a variable as any of the data types available for columns in a 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, 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; they 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;
```

Using Subscripts with Variables

You can use subscripts with variables that have CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE, or TEXT data types, just as you can with SQL column names. The subscripts indicate the starting and ending character positions of the variable. Subscripts must always be constants. You cannot use variables as subscripts. The following example illustrates the usage:

```
DEFINE name CHAR(15);
LET name[4,7] = 'Ream';
SELECT fname[1,3] INTO name[1,3] FROM customer
    WHERE lname = 'Ream';
```

The portion of the variable contents that is delimited by the two subscripts is referred to as a substring.

Scope of Variables

A variable is valid within the statement block in which it is defined. It is valid within statement blocks that are 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 following procedure, 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 accessible again.

```
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 and 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 in a DEFINE statement.

In some cases, you must change the name of the variable. For example, you cannot define a variable with the name **count** or **max**, because they are the names of aggregate functions. For a list of the keywords that can be used ambiguously, see the Identifier segment in the *Informix Guide to SQL: Syntax*.

Variables and Column Names

If you use the same identifier for a procedural variable as you use for a column name, the database server assumes that each 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 in the block of code in which the variable is defined. The following example shows a block in a procedure in which the variable called **user** is defined. This definition disallows the use of the USER function 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 the *Informix Guide to SQL: Syntax*.

SPL 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 in the *Informix Guide to SQL: Syntax*.

The following examples contain SQL expressions:

```
var1
var1 + var2 + 5
read_address('Miller')
read_address(lastname = 'Miller')
get_duedate(acct_num) + 10 UNITS DAY
fname[1,5] || ''|| lname
'(415)' || get_phonenum(cust_name)
```

Assigning Values to Variables

You can assign a value to a procedure variable in the following ways:

- Use a LET statement.
- Use a SELECT...INTO statement.
- Use a CALL statement with a procedure that has a RETURNING clause.
- Use 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

Stored Procedure Language (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. The statements that control program flow are described briefly in this section. For the syntax and complete descriptions of these statements, see the *Informix Guide to SQL: Syntax*.

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 the IF statement includes an ELSE clause or ELIF (else if) clause. The following example shows 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

The following table shows the three statements you can use to accomplish looping in SPL.

Statement	Action
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.

The following table shows the statements you can use to leave a loop.

Statement	Action
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 EXCEPTION	Exits the loop if the exception is not trapped (caught) in the body of the loop.

For more information about the syntax and use of these statements, see the *Informix Guide to SQL: Syntax*.

Function Handling

You can call procedures as well as run operating-system commands from in a procedure.

Calling Procedures Within a Procedure

Use a CALL statement or the SQL statement EXECUTE PROCEDURE to execute a procedure from a procedure. The following example shows a call to the **read_name** procedure using 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 a Procedure

UNIX

Use the SYSTEM statement to execute a system call from a procedure. The following example shows a call to the **echo** command:

```
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
```

♦

WIN NT

Use the SYSTEM statement to execute a system call from a procedure. In the following example, the first SYSTEM statement causes the Windows NT operating system to send an error message to a temporary file and to put the message in a system log that is sorted alphabetically. The second SYSTEM statement in the example causes the operating system to delete the temporary file.

```
CREATE PROCEDURE test_proc()
    .
    .
    .
    SYSTEM 'type errormess101 > %tmp%tmpfile.txt |
            sort >> %SystemRoot%systemlog.txt';
    SYSTEM 'del %tmp%tmpfile.txt';
    .
    .
    .
END PROCEDURE; --test_proc
```

The expressions that follow the SYSTEM statements in this example contain two variables, **%tmp%** and **%SystemRoot%**. The Windows NT operating system defines both of these variables. ♦

Calling a Procedure Recursively

You can call a procedure from itself. No restrictions apply on calling a procedure recursively.

Passing Information to and from a Procedure

When you create a procedure, you specify an argument list to determine whether it expects information to be passed to it. 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 requires a single piece of integer information passed to it, you can provide a procedure heading as the following example shows:

```
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: one line to state the data types that are going to be returned, and one line to return the values explicitly.

Specifying Return Data Types

Immediately after you specify 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;
    DEFINE o_name CHAR(10);
    .
    .
    .
    RETURN ncalls, o_name;
    .
    .
    .
END PROCEDURE;
```

If you neglect to include a RETURN statement, you do not get an error message at compile time.

Returning More Than One Set of Values from a Procedure

If your procedure executes a SELECT statement 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.

The following example shows 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.

```
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 use the ON EXCEPTION statement to trap any exception (or error) that the database server returns to your procedure, or any exception raised by your procedure. The RAISE EXCEPTION statement lets you 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, you can take recovery action.

The following example shows 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. 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 the statement blocks nested within that following statement block, and all 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 the following example 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.

```
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 with the RAISE EXCEPTION statement, as the following pseudocode example shows. In this example, the ON EXCEPTION statement uses two variables, **esql** and **eisam**, to hold the error numbers that the database server returns. The IF clause executes if an error occurs and if the SQL error number is -206. If any other SQL error is trapped, it is passed out of this BEGIN...END block to the last BEGIN...END block of the previous example.

```
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 the following example shows. Here, if the user is **pault**, then the stored procedure acts as if that user has no update privileges, even if the user 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

The following example shows how you can use the RAISE EXCEPTION statement to break out of a deeply nested block. 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.

```
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 you set a trap for this error somewhere in the block, the error condition is passed back to the block that contains the call and back to any blocks that contain the block. If no ON EXCEPTION statement exists 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.

Summary

Stored procedures provide many opportunities for streamlining your database process, including enhanced database performance, simplified applications, and limited or monitored access to data. For syntax diagrams of SPL statements, see the *Informix Guide to SQL: Syntax*.

Chapter

Creating and Using Triggers

9

When to Use Triggers 9-3

How to Create a Trigger 9-4
- Assigning a Trigger Name 9-5
- Specifying the Trigger Event 9-5
- Defining the Triggered Actions 9-6
- A Complete CREATE TRIGGER Statement 9-7

Using Triggered Actions 9-7
- Using BEFORE and AFTER Triggered Actions 9-7
- Using FOR EACH ROW Triggered Actions 9-9
  - Using the REFERENCING Clause 9-9
  - Using the WHEN Condition 9-10
- Using Stored Procedures as Triggered Actions 9-11
  - Passing Data to a Stored Procedure 9-11
  - Using the Stored Procedure Language 9-12
  - Updating Nontriggering Columns with Data from a Stored Procedure. 9-12

Reentrant Triggers for Dynamic Server 9-13

Tracing Triggered Actions 9-13
- Example of TRACE Statements in a Stored Procedure 9-14
- Example of TRACE Output 9-14

Generating Error Messages 9-15
- Applying a Fixed Error Message 9-15
- Generating a Variable Error Message 9-17

Summary 9-18

An SQL trigger is a mechanism that resides in the database. It is available to any user who has permission to use it. It specifies that when a particular action, an insert, a delete, or an update, occurs on a particular table, the database server should automatically perform one or more additional actions. The additional actions can be INSERT, DELETE, UPDATE, or EXECUTE PROCEDURE statements.

This chapter describes the purpose of each component of the CREATE TRIGGER statement, illustrates some uses for triggers, and describes the advantages of using a stored procedure as a triggered action.

AD/XP

Informix Dynamic Server with Advanced Decision Support and Extended Parallel Options does not support SQL triggers. ♦

When to Use Triggers

Because a trigger resides in the database and anyone who has the required privilege can use it, a trigger lets you write a set of SQL statements that multiple applications can use. It lets you avoid redundant code when multiple programs need to perform the same database operation.

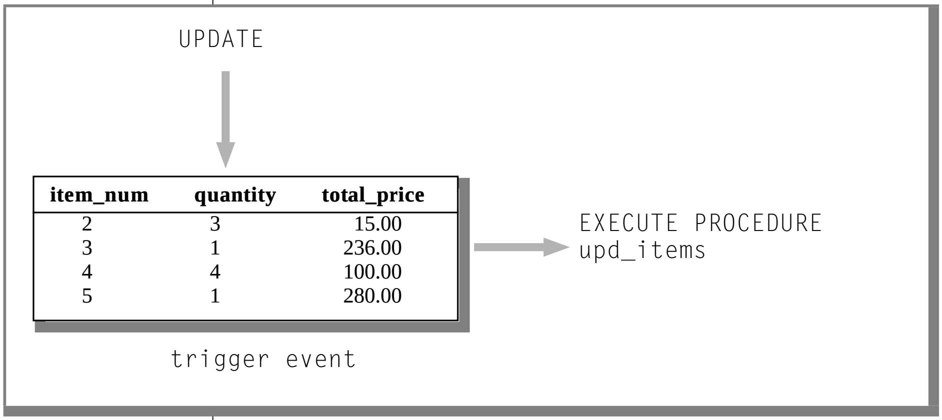
You can use triggers to perform the following actions as well as others that are not found in this list:

- Create an audit trail of activity in the database. For example, you can track updates to the orders table by updating corroborating information to an audit table.
- Implement a business rule. For example, you can determine when an order exceeds a customer's credit limit and display a message to that effect.

- Derive additional data that is not available within a table or within the database. For example, when an update occurs to the quantity column of the **items** table, you can calculate the corresponding adjustment to the **total_price** column.
- Enforce referential integrity. When you delete a customer, for example, you can use a trigger to delete corresponding rows (that is, rows that have the same customer number) in the **orders** table.

How to Create a Trigger

You use the CREATE TRIGGER statement to create a trigger. The CREATE TRIGGER statement is a data-definition statement that associates SQL statements with a precipitating action on a table. When the precipitating action occurs, it triggers the associated SQL statements, which are stored in the database. Figure 9-1 illustrates the relationship of the precipitating action, or trigger event, to the triggered action.



item_num	quantity	total_price
2	3	15.00
3	1	236.00
4	4	100.00
5	1	280.00

Figure 9-1
Trigger Event and Triggered Action

The CREATE TRIGGER statement consists of clauses that perform the following actions:

- Assign a trigger name.
- Specify the trigger event, that is, the table and the type of action that initiate the trigger.
- Define the SQL actions that are triggered.

An optional clause, called the REFERENCING clause is discussed in "Using FOR EACH ROW Triggered Actions" on page 9-9.

To create a trigger, you can use DB-Access, Relational Object Manager, or one of the SQL APIs. This section describes the CREATE TRIGGER statement as you enter it with the interactive Query-language option in DB-Access or Relational Object Manager. In an SQL API, you precede the statement with the symbol or keywords that identify it as an embedded statement.

Assigning a Trigger Name

The trigger name identifies the trigger. It follows the words CREATE TRIGGER in the statement. It can be up to 18 characters in length, beginning with a letter and consisting of letters, the digits 0 to 9, and the underscore. In the following example, the portion of the CREATE TRIGGER statement that is shown assigns the name **upqty** to the trigger:

```
CREATE TRIGGER upqty    -- assign trigger name
```

Specifying the Trigger Event

The *trigger event* is the type of statement that activates the trigger. When a statement of this type is performed on the table, the database server executes the SQL statements that make up the triggered action. The trigger event can be an INSERT, DELETE, or UPDATE statement. When you define an UPDATE trigger event, you can name one or more columns in the table to activate the trigger. If you do not name any columns, then an update of any column in the table activates the trigger. You can create only one INSERT and one DELETE trigger per table, but you can create multiple UPDATE triggers as long as the triggering columns are mutually exclusive.

In the following excerpt of a CREATE TRIGGER statement, the trigger event is defined as an update of the **quantity** column in the **items** table:

```
CREATE TRIGGER upqty
UPDATE OF quantity ON items-- an UPDATE trigger event
```

This portion of the statement identifies the table on which you create the trigger. If the trigger event is an insert or delete, only the type of statement and the table name are required, as the following example shows:

```
CREATE TRIGGER ins_qty
INSERT ON items          -- an INSERT trigger event
```

Defining the Triggered Actions

The *triggered actions* are the SQL statements that are performed when the trigger event occurs. The triggered actions can consist of INSERT, DELETE, UPDATE, and EXECUTE PROCEDURE statements. In addition to specifying what actions are to be performed, however, you must also specify *when* they are to be performed in relation to the triggering statement. You have the following choices:

- Before the triggering statement executes
- After the triggering statement executes
- For each row that is affected by the triggering statement

A single trigger can define actions for each of these times.

To define a triggered action, you specify when it occurs and then provide the SQL statement or statements to execute. You specify when the action is to occur with the keywords BEFORE, AFTER, or FOR EACH ROW. The triggered actions follow, enclosed in parentheses. The following triggered-action definition specifies that the stored procedure **upd_items_p1** is to be executed before the triggering statement:

```
BEFORE(EXECUTE PROCEDURE upd_items_p1)-- a BEFORE action
```

A Complete CREATE TRIGGER Statement

To define a complete CREATE TRIGGER statement, you combine the trigger-name clause, the trigger-event clause, and the triggered-action clause. The following CREATE TRIGGER statement is the result of combining the components of the statement from the preceding examples. This trigger executes the stored procedure **upd_items_p1** whenever the **quantity** column of the **items** table is updated.

```
CREATE TRIGGER upqty
UPDATE OF quantity ON items
BEFORE(EXECUTE PROCEDURE upd_items_p1)
```

If a database object in the trigger definition, such as the stored procedure **upd_items_p1** in this example, does not exist when the database server processes the CREATE TRIGGER statement, it returns an error.

Using Triggered Actions

To use triggers effectively, you need to understand the relationship between the triggering statement and the resulting triggered actions. You define this relationship when you specify the time that the triggered action occurs; that is, BEFORE, AFTER, or FOR EACH ROW.

Using BEFORE and AFTER Triggered Actions

Triggered actions that occur before or after the trigger event execute only once. A BEFORE triggered action executes before the *triggering statemen*t, that is, before the occurrence of the trigger event. An AFTER triggered action executes after the action of the triggering statement is complete. BEFORE and AFTER triggered actions execute even if the triggering statement does not process any rows.

Among other uses, you can use BEFORE and AFTER triggered actions to determine the effect of the triggering statement. For example, before you update the **quantity** column in the **items** table, you could call the stored procedure **upd_items_p1** to calculate the total quantity on order for all items in the table, as the following example shows. The procedure stores the total in a global variable called **old_qty**.

```
CREATE PROCEDURE upd_items_p1()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    LET old_qty = (SELECT SUM(quantity) FROM items);
END PROCEDURE;
```

After the triggering update completes, you can calculate the total again to see how much it has changed. The following stored procedure, **upd_items_p2**, calculates the total of **quantity** again and stores the result in the local variable **new_qty**. Then it compares **new_qty** to the global variable **old_qty** to see if the total quantity for all orders has increased by more than 50 percent. If so, the procedure uses the RAISE EXCEPTION statement to simulate an SQL error.

```
CREATE PROCEDURE upd_items_p2()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    DEFINE new_qty INT;
    LET new_qty = (SELECT SUM(quantity) FROM items);
    IF new_qty > old_qty * 1.50 THEN
        RAISE EXCEPTION -746, 0, 'Not allowed - rule violation';
    END IF
END PROCEDURE;
```

The following trigger calls **upd_items_p1** and **upd_items_p2** to prevent an extraordinary update on the **quantity** column of the **items** table:

```
CREATE TRIGGER up_items
UPDATE OF quantity ON items
BEFORE(EXECUTE PROCEDURE upd_items_p1())
AFTER(EXECUTE PROCEDURE upd_items_p2());
```

If an update raises the total quantity on order for all items by more than 50 percent, the RAISE EXCEPTION statement in **upd_items_p2** terminates the trigger with an error. When a trigger fails in the database server and the database has logging, the database server rolls back the changes that both the triggering statement and the triggered actions make. For more information on what happens when a trigger fails, see the CREATE TRIGGER statement in the *Informix Guide to SQL: Syntax*.

Using FOR EACH ROW Triggered Actions

A FOR EACH ROW triggered action executes once for each row that the triggering statement affects. For example, if the triggering statement has the following syntax, a FOR EACH ROW triggered action executes once for each row in the **items** table in which the **manu_code** column has a value of 'KAR':

```
UPDATE items SET quantity = quantity * 2 WHERE manu_code = 'KAR'
```

If the triggering statement does not process any rows, a FOR EACH ROW triggered action does not execute.

Using the REFERENCING Clause

When you create a FOR EACH ROW triggered action, you must usually indicate in the triggered action statements whether you are referring to the value of a column before or after the effect of the triggering statement. For example, imagine that you want to track updates to the **quantity** column of the **items** table. To do this, you create the following table to record the activity:

```
CREATE TABLE log_record
    (item_num    SMALLINT,
    ord_num      INTEGER,
    username     CHARACTER(8),
    update_time  DATETIME YEAR TO MINUTE,
    old_qty      SMALLINT,
    new_qty      SMALLINT);
```

To supply values for the **old_qty** and **new_qty** columns in this table, you must be able to refer to the old and new values of **quantity** in the **items** table; that is, the values before and after the effect of the triggering statement. The REFERENCING clause enables you to do this.

The REFERENCING clause lets you create two prefixes that you can combine with a column name, one to reference the old value of the column and one to reference its new value. These prefixes are called *correlation names*. You can create one or both correlation names, depending on your requirements. You indicate which one you are creating with the keywords OLD and NEW. The following REFERENCING clause creates the correlation names **pre_upd** and **post_upd** to refer to the old and new values in a row:

```
REFERENCING OLD AS pre_upd NEW AS post_upd
```

The following triggered action creates a row in **log_record** when **quantity** is updated in a row of the **items** table. The INSERT statement refers to the old values of the **item_num** and **order_num** columns and to both the old and new values of the **quantity** column.

```
FOR EACH ROW(INSERT INTO log_record
    VALUES (pre_upd.item_num, pre_upd.order_num, USER, CURRENT,
            pre_upd.quantity, post_upd.quantity));
```

The correlation names defined in the REFERENCING clause apply to all rows affected by the triggering statement.

***Important:** If you refer to a column name in the triggering table and do not use a correlation name, the database server makes no special effort to search for the column in the definition of the triggering table. You must always use a correlation name with a column name in SQL statements in a FOR EACH ROW triggered action, unless the statement is valid independent of the triggered action. For more information, see the CREATE TRIGGER statement in the Informix Guide to SQL: Syntax.*

Using the WHEN Condition

As an option, you can precede a triggered action with a WHEN clause to make the action dependent on the outcome of a test. The WHEN clause consists of the keyword WHEN followed by the condition statement given in parentheses. In the CREATE TRIGGER statement, the WHEN clause follows the keywords BEFORE, AFTER, or FOR EACH ROW and precedes the triggered-action list.

When a WHEN condition is present, if it evaluates to *true*, the triggered actions execute in the order in which they appear. If the WHEN condition evaluates to *false* or *unknown*, the actions in the triggered-action list do not execute. If the trigger specifies FOR EACH ROW, the condition is evaluated for each row also.

In the following trigger example, the triggered action executes only if the condition in the WHEN clause is true; that is, if the post-update unit price is greater than two times the pre-update unit price:

```
CREATE TRIGGER up_price
UPDATE OF unit_price ON stock
REFERENCING OLD AS pre NEW AS post
FOR EACH ROW WHEN(post.unit_price > pre.unit_price * 2)
    (INSERT INTO warn_tab VALUES(pre.stock_num, pre.manu_code,
        pre.unit_price, post.unit_price, CURRENT))
```

For more information on the WHEN condition, see the CREATE TRIGGER statement in the *Informix Guide to SQL: Syntax*

Using Stored Procedures as Triggered Actions

Probably the most powerful feature of triggers is the ability to call a stored procedure as a triggered action. The EXECUTE PROCEDURE statement, which calls a stored procedure, lets you pass data from the triggering table to the stored procedure and also to update the triggering table with data returned by the stored procedure. SPL also lets you define variables, assign data to them, make comparisons, and use procedural statements to accomplish complex tasks within a triggered action.

Passing Data to a Stored Procedure

You can pass data to a stored procedure in the argument list of the EXECUTE PROCEDURE statement. The EXECUTE PROCEDURE statement in the following trigger example passes values from the **quantity** and **total_price** columns of the **items** table to the stored procedure **calc_totpr**:

```
CREATE TRIGGER upd_totpr
UPDATE OF quantity ON items
REFERENCING OLD AS pre_upd NEW AS post_upd
FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
    post_upd.quantity, pre_upd.total_price) INTO total_price)
```

Passing data to a stored procedure lets you use it in the operations that the procedure performs.

Using the Stored Procedure Language

The EXECUTE PROCEDURE statement in the preceding trigger calls the stored procedure that the following example shows. The procedure uses SPL to calculate the change that needs to be made to the **total_price** column when **quantity** is updated in the **items** table. The procedure receives both the old and new values of **quantity** and the old value of **total_price**. It divides the old total price by the old quantity to derive the unit price. It then multiplies the unit price by the new quantity to obtain the new total price.

```
CREATE PROCEDURE calc_totpr(old_qty SMALLINT, new_qty SMALLINT,
    total MONEY(8)) RETURNING MONEY(8);
    DEFINE u_price LIKE items.total_price;
    DEFINE n_total LIKE items.total_price;
    LET u_price = total / old_qty;
    LET n_total = new_qty * u_price;
    RETURN n_total;
END PROCEDURE;
```

In this example, SPL lets the trigger derive data that is not directly available from the triggering table.

Updating Nontriggering Columns with Data from a Stored Procedure

Within a triggered action, the INTO clause of the EXECUTE PROCEDURE statement lets you update nontriggering columns in the triggering table. The EXECUTE PROCEDURE statement in the following example calls the **calc_totpr** stored procedure that contains an INTO clause, which references the column **total_price**:

```
FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
    post_upd.quantity, pre_upd.total_price) INTO total_price);
```

The value that is updated into **total_price** is returned by the RETURN statement at the conclusion of the stored procedure. The **total_price** column is updated for each row that the triggering statement affects.

IDS

Reentrant Triggers for Dynamic Server

Dynamic Server supports reentrant triggers. A *reentrant trigger* refers to a case in which the triggered action can reference the triggering table. In other words, both the triggering event and the triggered action can operate on the same table. For example, suppose the following UPDATE statement represents the triggering event:

```
UPDATE tab1 SET (col_a, col_b) = (col_a + 1, col_b + 1)
```

The following triggered action is legal because column **col_c** is not a column that the triggering event has updated:

```
UPDATE tab1 SET (col_c) = (col_c + 3)
```

In the preceding example, a triggered action on **col_a** or **col_b** would be illegal because a triggered action cannot be an UPDATE statement that references a column that was updated by the triggering event.

For a list of the rules that describe all situations in which a trigger can and cannot be reentrant, see the CREATE TRIGGER statement in the *Informix Guide to SQL: Syntax*.

Tracing Triggered Actions

If a triggered action does not behave as you expect, place it in a stored procedure, and use the SPL TRACE statement to monitor its operation. Before you start the trace, you must direct the output to a file with the SET DEBUG FILE TO statement.

Example of TRACE Statements in a Stored Procedure

The following example shows TRACE statements that you add to the stored procedure **items_pct.** The SET DEBUG FILE TO statement directs the trace output to the file that the pathname specifies. The TRACE ON statement begins tracing the statements and variables within the procedure.

```
CREATE PROCEDURE items_pct(mac CHAR(3))
DEFINE tp MONEY;
DEFINE mc_tot MONEY;
DEFINE pct DECIMAL;
SET DEBUG FILE TO 'pathname'; -- modify this pathname according to the
                    -- conventions that your operating system requires

TRACE 'begin trace';
TRACE ON;
LET tp = (SELECT SUM(total_price) FROM items);
LET mc_tot = (SELECT SUM(total_price) FROM items
    WHERE manu_code = mac);
LET pct = mc_tot / tp;
IF pct > .10 THEN
    RAISE EXCEPTION -745;
END IF
TRACE OFF;
END PROCEDURE;

CREATE TRIGGER items_ins
INSERT ON items
REFERENCING NEW AS post_ins
FOR EACH ROW(EXECUTE PROCEDURE items_pct (post_ins.manu_code));
```

Example of TRACE Output

The following example shows sample trace output from the **items_pct** procedure as it appears in the file that was named in the SET DEBUG FILE TO statement. The output reveals the values of procedure variables, procedure arguments, return values, and error codes.

```
trace expression :begin trace
trace on
expression:
  (select (sum total_price)
    from items)
evaluates to $18280.77 ;
let  tp = $18280.77
expression:
  (select (sum total_price)
    from items
    where (= manu_code, mac))
evaluates to $3008.00 ;
let  mc_tot = $3008.00
expression:(/ mc_tot, tp)
```

```
evaluates to 0.16
let  pct = 0.16
expression:(> pct, 0.1)
evaluates to 1
expression:(- 745)
evaluates to -745
raise exception :-745, 0, ''
exception : looking for handler
SQL error = -745 ISAM error = 0  error string =  = ''
exception : no appropriate handler
```

For more information about how to use the TRACE statement to diagnose logic errors in stored procedures, see "Creating and Using Stored Procedures."

Generating Error Messages

When a trigger fails because of an SQL statement, the database server returns the SQL error number that applies to the specific cause of the failure.

When the triggered action is a stored procedure, you can generate error messages for other error conditions with one of two reserved error numbers. The first one is error number -745, which has a generalized and fixed error message. The second one is error number -746, which allows you to supply the message text, up to a maximum of 71 characters.

Applying a Fixed Error Message

You can apply error number -745 to any trigger failure that is not an SQL error. The following fixed message is for this error:

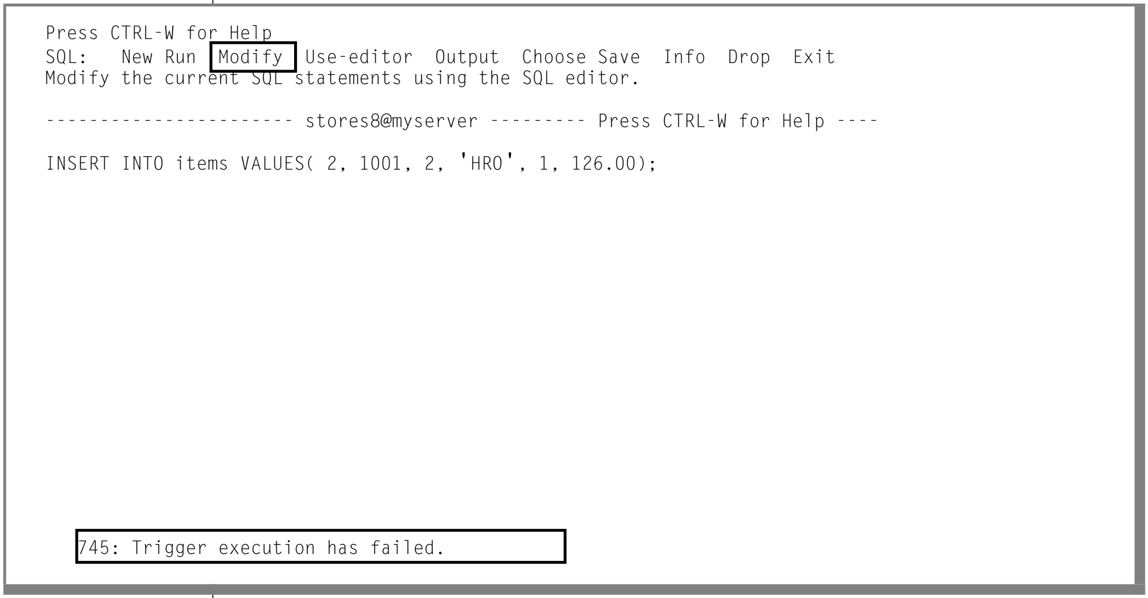
```
-745 Trigger execution has failed.
```

You can apply this message with the RAISE EXCEPTION statement in SPL. The following example generates error -745 if **new_qty** is greater than **old_qty** multiplied by 1.50:

```
CREATE PROCEDURE upd_items_p2()
    DEFINE GLOBAL old_qty INT DEFAULT 0;
    DEFINE new_qty INT;
    LET new_qty = (SELECT SUM(quantity) FROM items);
    IF new_qty > old_qty * 1.50 THEN
        RAISE EXCEPTION -745;
    END IF
END PROCEDURE
```

If you are using DB-Access, the text of the message for error -745 displays on the bottom of the screen, as seen in Figure 9-2.

Figure 9-2
Error Message -745 with Fixed Message



```
Press CTRL-W for Help
SQL:   New Run  Modify  Use-editor  Output  Choose Save  Info  Drop  Exit
Modify the current SQL statements using the SQL editor.

----------------------- stores8@myserver --------- Press CTRL-W for Help ----

INSERT INTO items VALUES( 2, 1001, 2, 'HRO', 1, 126.00);










    745: Trigger execution has failed.
```

If you trigger the erring procedure through an SQL statement in your SQL API, the database server sets the SQL error status variable to -745 and returns it to your program. To display the text of the message, follow the procedure that your Informix application development tool provides for retrieving the text of an SQL error message.

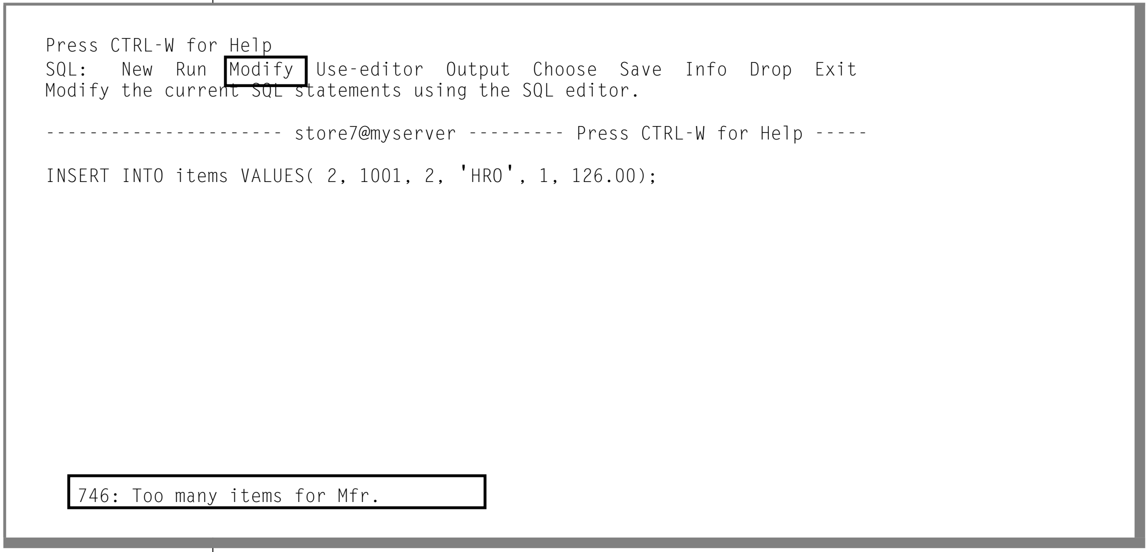
Generating a Variable Error Message

Error number -746 allows you to provide the text of the error message. Like the preceding example, the following one also generates an error if **new_qty** is greater than **old_qty** multiplied by 1.50. However, in this case the error number is -746, and the message text `Too many items for Mfr.` is supplied as the third argument in the RAISE EXCEPTION statement. For more information on the syntax and use of this statement, see the RAISE EXCEPTION statement in "Creating and Using Stored Procedures."

```
CREATE PROCEDURE upd_items_p2()
   DEFINE GLOBAL old_qty INT DEFAULT 0;
   DEFINE new_qty INT;
   LET new_qty = (SELECT SUM(quantity) FROM items);
   IF new_qty > old_qty * 1.50 THEN
      RAISE EXCEPTION -746, 0, 'Too many items for Mfr.';
   END IF
END PROCEDURE;
```

If you use DB-Access to submit the triggering statement, and if **new_qty** is greater than **old_qty**, you will get the result that Figure 9-3 shows.

Figure 9-3
Error Number -746 with User-Specified Message Text



```
Press CTRL-W for Help
SQL:   New  Run  Modify  Use-editor  Output  Choose  Save  Info  Drop  Exit
Modify the current SQL statements using the SQL editor.

---------------------- store7@myserver --------- Press CTRL-W for Help -----

INSERT INTO items VALUES( 2, 1001, 2, 'HRO', 1, 126.00);



746: Too many items for Mfr.
```

If you invoke the trigger through an SQL statement in an SQL API, the database server sets **sqlcode** to `-746` and returns the message text in the **sqlerrm** field of the SQL communications area (SQLCA). For complete information about how to use the SQLCA, see your SQL API manual.

Summary

To introduce triggers, this chapter discusses the following topics:

- The purpose of each component of the CREATE TRIGGER statement
- How to create BEFORE and AFTER triggered actions and how to use them to determine the impact of the triggering statement
- How to create a FOR EACH ROW triggered action and how to use the REFERENCING clause to refer to the values of columns both before and after the action of the triggering statement
- The advantages of using stored procedures as triggered actions
- How to trace triggered actions if they behave unexpectedly
- How to generate two types of error messages within a triggered action

Index

A

Access modes, description of 7-18
Accessing tables 2-101
Active set
  definition of 2-28
  of a cursor 5-24
Aggregate function
  and GROUP BY clause 3-5
  AVG 2-59
  COUNT 2-58
  description of 2-57, 2-68
  in ESQL 5-14
  in SPL expressions 8-25
  in subquery 3-33
  MAX 2-60
  mentioned Intro-6
  MIN 2-60
  null value signalled 5-12
  RANGE 2-60
  standard deviation 2-61
  STDEV 2-61
  SUM 2-60
  VARIANCE 2-62
Alias
  for table name 2-96
  to assign column names in temporary table 3-12
  with self-join 3-11
ALL keyword, beginning a subquery 3-31
ALTER INDEX statement, locking table 7-8
AND logical operator 2-36
ANSI 1-16
ANSI compliance
  icon Intro-11
  level Intro-15
ANSI-compliant database
  FOR UPDATE not required in 6-16
  signalled in SQLWARN 5-11
ANY keyword, beginning a subquery 3-32
Application
  common features 1-17
  description of 1-18
  design of order-entry 4-29
  handling errors 5-17
Archiving
  database server methods 4-33
  description of 4-33
  transaction log 4-33
Arithmetic operator, in expression 2-49
Ascending order in SELECT 2-14
Asterisk, wildcard character in SELECT 2-12
AVG function, as aggregate function 2-59

B

BEGIN WORK statement 4-32
BETWEEN keyword
  testing for equality in WHERE clause 2-29
  using to specify a range of rows 2-32
Boolean expression, and logical operator 2-36

BYTE data type
  restrictions
    with GROUP BY 3-6
    with LIKE or MATCHES 2-37
    with relational expression 2-29
  using LENGTH function on 2-80
BYTE value, displaying 2-11

C

CALL statement
  assigning values with 8-25
  executing a procedure 8-12
Cartesian product
  basis of any join 2-90
  description of 2-88
Cascading deletes
  definition of 4-24
  locking associated with 4-24
  logging 4-24
  restriction 4-26
Case conversion
  with INITCAP function 2-73
  with LOWER function 2-72
  with UPPER function 2-72
CASE expression
  description of 2-54
  mentioned Intro-7
  using 2-55
CHAR data type
  converting to a DATE value 2-68
  converting to a DATETIME value 2-70
  in relational expressions 2-29
  subscripting 2-44
  substrings of 2-27
  truncation signalled 5-12
Character string
  converting to a DATE value 2-68
  converting to a DATETIME value 2-70
Check constraint, definition of 4-22
CLOSE DATABASE statement, effect on database locks 7-8
Codd, E. F. 1-11
Collation order and GLS 2-25
Column
  definition of 2-5
  description of 1-13
  in relational model 1-13
  label on 3-44
Column number, using 2-24
Comment icons Intro-9
COMMIT WORK statement
  closing cursors 7-21
  releasing locks 7-11, 7-21
  setting SQLCODE 6-5
Committed Read, isolation level (Informix) 7-14
Comparison condition, description of 2-29
Compliance
  icons Intro-11
  with industry standards Intro-15
Compound query 3-40
Concurrency
  access modes 7-18
  ANSI Read Committed isolation 7-14
  ANSI Read Uncommitted isolation 7-14
  ANSI Repeatable Read isolation 7-17
  ANSI Serializable isolation 7-17
  database lock 7-8
  deadlock 7-20
  description of 4-34, 7-3
  effect on performance 7-3
  Informix Cursor Stability isolation 7-15
  Informix Dirty Read isolation 7-14
  Informix Read Committed isolation 7-14
  Informix Repeatable Read isolation 7-17
  isolation level 7-12
  kinds of locks 7-7
  lock duration 7-11
  lock scope 7-7
  table lock 7-8
Configuration parameter, ISOLATION_LOCKS 7-15
Constraint
  disabled 4-27
  enabled 4-27
  entity integrity 4-21
CONTINUE statement, exiting a loop 8-27
Conversion function, description of 2-68
Coordinated deletes 6-6
Correlated subquery
  definition of 3-30
  restriction with cascading deletes 4-26
COUNT function
  and GROUP BY 3-6
  as aggregate function 2-58
  count rows to delete 4-5
  use in a subquery 4-6
  with DISTINCT 2-59
CREATE DATABASE statement
  setting shared lock 7-8
  SQLWARN after 5-11
CREATE INDEX statement, locking table 7-8
CREATE PROCEDURE FROM statement, in embedded languages 8-7
CREATE PROCEDURE statement
  inside CREATE PROCEDURE FROM 8-7
  using 8-6
CURRENT function, comparing column values 2-63
Cursor
  active set of 5-24
  closing 7-21
  declaring 5-20
  defined 5-24
  definition of 5-20
  for insert 6-9
  for update 6-15, 7-11
  hold 7-21
  opening 5-21, 5-24
  retrieving values with FETCH 5-22
  scroll 5-23
  sequence of program operations 5-20

sequential 5-23, 5-25
WITH HOLD 7-21
Cursor Stability isolation level (Informix) Intro-7, 7-15
Cyclic query 4-26

D

Data definition statements 5-34
Data integrity 4-28 to 4-33
Data loading 4-34
Data model, description of 1-4
Data replication 4-35
Data type
automatic conversions 5-15
conversion 4-9
in SPL variables 8-22
Database
ANSI-compliant 1-17
application 1-17
concurrent use 1-9
definition of 1-12
GLS 1-17
management of 1-10
relational, definition of 1-11
server 1-9, 1-17
server, definition of 1-18
table names 2-101
Database lock 7-8
Database object
constraints as a 4-26
index as a 4-26
object modes 4-26
trigger as a 4-26
violation detection 4-26
Database server
archiving 4-33
definition of 1-18
identifying host computer name 2-83
identifying version number 2-83
locking tables 7-8
signalled in SQLWARN 5-11
DATABASE statement
exclusive mode 7-8
locking 7-8
SQLWARN after 5-11
Database-level privilege 4-17
DATE data type
converting to a character string 2-69
functions returning 2-63
in ORDER BY sequence 2-14
in relational expressions 2-29
international date formats 1-17
DATE function, as conversion function 2-68
DATETIME data type
converting to a character string 2-69
displaying format 2-67
functions returning 2-63
in ORDER BY sequence 2-14
in relational expressions 2-29
DAY function, as time function 2-64
DB-Access, creating database with 5-34
DBA-privileged procedure 8-17
DBDATE environment variable 4-9
DBINFO function, in SELECT statement 2-82
DBSERVERNAME function, in SELECT statement 2-81, 3-19
dbspace, name returned by DBINFO function 2-82
Deadlock detection 7-20
DECIMAL data type, signalled in SQLWARN 5-11
DECLARE statement
description of 5-20
FOR INSERT clause 6-9
FOR UPDATE 6-15
SCROLL keyword 5-24
WITH HOLD clause 7-23
DECODE function 2-83
Default locale Intro-4
Default value, in column 4-22
DEFINE statement, in stored procedures 8-21
DELETE statement
all rows of table 4-4
coordinated deletes 6-6
count of rows 6-4
description of 4-4
embedded 5-6, 6-3 to 6-8
number of rows 5-10
preparing 5-31
transactions with 6-5
using subquery 4-6
WHERE clause restricted 4-7
with cursor 6-7
Demonstration database Intro-5
Descending order in SELECT 2-14
Diagnostics table, using with filtering mode 4-27
Difference set operation 3-50
Dirty Read isolation level (Informix) 7-14
Disabled object mode, definition of 4-26
Display label
in ORDER BY clause 2-56
with SELECT 2-52
DISTINCT keyword
relation to GROUP BY 3-4
using in SELECT 2-20
using with COUNT function 2-59
Distributed deadlock 7-21
DOCUMENT keyword, use in stored procedures 8-8
Documentation conventions
icon Intro-9
sample-code Intro-11
typographical Intro-8
Documentation notes, program item Intro-14
Documentation, types of
documentation notes Intro-14
error message files Intro-13
machine notes Intro-14
on-line manuals Intro-12
printed manuals Intro-13
related reading Intro-15
release notes Intro-14
Domain of column 4-21
Dominant table 3-20
DROP INDEX statement, locking table 7-8
Duplicate values, finding 3-15
Dynamic SQL
description of 5-5, 5-29
freeing prepared statements 5-33

E

Embedded SQL
  definition of 5-4
  languages available 5-4
Enabled object mode, definition of 4-26
End of data
  signal in SQLCODE 5-9, 5-17
  signal only for SELECT 6-14
  when opening cursor 5-21
Entity, definition of 4-21
en_us.8859-1 locale Intro-4
Equals (=) relational operator 2-30, 2-90
Equi-join 2-90
Error checking
  exception handling 8-32
  in stored procedures 8-32
  simulating errors 8-34
Error message files Intro-13
Error message variable 5-13
Error messages
  for trigger failure 9-15
  generating in a trigger 9-15
  retrieving trigger text in a program 9-16, 9-18
Errors
  after DELETE 6-4
  at compile time 8-8
  codes for 5-10
  dealing with 5-17
  detected on opening cursor 5-21
  during updates 4-28
  in stored procedure syntax 8-9
  inserting with a cursor 6-11
  ISAM error code 5-10
ESCAPE keyword, using in WHERE clause 2-44
ESQL
  cursor use 5-20 to 5-28
  DELETE statement in 6-3
  delimiting host variables 5-7
  dynamic embedding 5-5, 5-29
  error handling 5-17
  fetching rows from cursor 5-22
  host variable 5-6, 5-8
  indicator variable 5-16
  INSERT in 6-9
  overview 5-3 to 5-37, 6-3 to 6-18
  preprocessor 5-4
  scroll cursor 5-23
  selecting single rows 5-14
  SQL Communications Area (SQLCA) 5-8
  SQLCODE 5-9
  SQLERRD fields 5-10
  static embedding 5-5
  UPDATE in 6-15
Exclusive lock 7-7
EXECUTE IMMEDIATE statement, description of 5-33
EXECUTE PROCEDURE statement
  assigning values with 8-25
  using 8-12
EXECUTE statement, description of 5-31
EXISTS keyword, in a WHERE clause 3-31
EXIT statement, exiting a loop 8-27
Expression
  CASE 2-54
  date-oriented 2-63
  description of 2-49
  display label for 2-52
  SPL 8-25
EXTEND function
  using in expression 2-67
  with DATE, DATETIME and INTERVAL 2-63
Extension, to SQL, symbol for Intro-11
External tables Intro-7, 4-34

F

Feature icons Intro-10
Features, product Intro-5
FETCH statement
  ABSOLUTE keyword 5-24
  description of 5-22
  sequential 5-23
  with sequential cursor 5-25
File, compared to database 1-4
Filtering object mode, definition of 4-26
finderr utility Intro-13
FIRST clause
  description of 2-46
  in a union query 2-48
  using 2-46
  with ORDER BY clause 2-47
FLUSH statement
  count of rows inserted 6-11
  writing rows to buffer 6-10
FOR statement, looping in a stored procedure 8-27
FOR UPDATE keywords
  conflicts with ORDER BY 6-8
  not needed in ANSI-compliant database 6-16
  specific columns 6-16
FOREACH statement, looping in a stored procedure 8-27
Foreign key 4-22
Fragmented table, using primary keys 3-15
FREE statement, freeing prepared statements 5-33
FROM keyword, alias names 2-96
Function
  aggregate 2-57
  applying to expressions 2-63
  conversion 2-68
  DATE 2-68
  date-oriented 2-63
  DBINFO 2-82
  DECODE 2-83
  in a stored procedure 8-28
  in SELECT statements 2-57
  INITCAP 2-73
  LOWER 2-72
  LPAD 2-77
  name confusion in SPL 8-24
  NVL 2-85
  REPLACE 2-74
  RPAD 2-78
  string manipulation 2-71
  SUBSTR 2-76
  SUBSTRING 2-75
  time 2-63
  TO_CHAR 2-69
  TO_DATE 2-70
  UPPER 2-72

G

Global Language Support (GLS)
- and MATCHES keyword 2-42
- and ORDER BY keywords 2-25, 2-42
- database, description of 1-17
- default locale 2-25
- locales Intro-4
- mentioned Intro-6
- sort order 2-25

Global variable in SPL 8-21
GLS. *See* Global Language Support.
GRANT statement, in embedded SQL 5-34 to 5-36
Greater than or equal to (>=) relational operator 2-32
Greater than (>) relational operator 2-31
GROUP BY keywords
- column number with 3-7
- description of 3-4

H

HAVING keyword 3-8
Header, of a procedure 8-30
HEX function, using in expression 2-82
Hold cursor, definition of 7-21
Host variable
- delimiter for 5-7
- description of 5-6
- fetching data into 5-22
- in DELETE statement 6-4
- in INSERT statement 6-9
- in UPDATE statement 6-15
- in WHERE clause 5-15
- INTO keyword sets 5-14
- null indicator 5-16
- restrictions in prepared statement 5-30
- truncation signalled 5-12
- with EXECUTE statement 5-31

Hostname of computer, returned by DBINFO function 2-83

I

Icons
- comment Intro-9
- compliance Intro-11
- feature Intro-10
- platform Intro-10
- product Intro-10

IF statement, branching 8-26
IN keyword, using in WHERE clause 2-29
IN relational operator 3-31
Index
- disabled mode 4-27
- enabled mode 4-27
- filtering mode 4-27
- table locks 7-8

Indicator variable, definition of 5-16
Industry standards, compliance with Intro-15
INFORMIXDIR/bin directory Intro-5
INITCAP function, as string manipulation function 2-73
Insert cursor
- definition of 6-9
- using 6-12

INSERT statement
- and end of data 6-14
- constant data with 6-12
- count of rows inserted 6-11
- duplicate values in 4-8
- embedded 6-9 to 6-14
- inserting
  - multiple rows 4-10
  - rows 4-7
  - single rows 4-7
- null values in 4-8
- number of rows 5-10
- SELECT statement in 4-10
- VALUES clause 4-7

Inserting rows of constant data 6-12
Interrupted modifications 4-28
Intersection set operation 3-48
INTERVAL data type, in relational expressions 2-29
INTO keyword
- choice of location 5-23
- in FETCH statement 5-23
- mismatch signalled in SQLWARN 5-12
- restrictions in INSERT 4-11
- restrictions in prepared statement 5-30
- retrieving multiple rows 5-20
- retrieving single rows 5-14

INTO TEMP keywords, description of 2-100
IS NOT NULL keywords 2-35
IS NULL keywords 2-35
ISAM error code 5-10
ISO 8859-1 code set Intro-4, 2-25
Isolation level
- ANSI Read Committed 7-14
- ANSI Read Uncommitted 7-14
- ANSI Repeatable Read 7-17
- ANSI Serializable 7-17
- description of 7-12
- Informix Committed Read 7-14
- Informix Cursor Stability 7-15
- Informix Dirty Read 7-14
- Informix Repeatable Read 7-17
- setting 7-12

ISOLATION_LOCKS configuration parameter, specifying number of rows to lock 7-15

J

Join
- associative 2-94
- composite 2-88
- condition 2-88
- creating 2-90
- definition of 2-8, 2-88
- dominant table 3-20
- equi-join 2-90
- in an UPDATE statement 4-17
- multiple-table join 2-95
- natural 2-93
- nested simple 3-24
- outer, definition of 3-20
- outer, types of 3-20

self-join 3-11
simple 2-88
subservient table 3-20

K

Key lock 7-10
Keywords
in a subquery 3-31
in a WHERE clause 2-29

L

Label 2-52, 3-44
Last SERIAL value, returned by DBINFO function 2-82
LENGTH function
on TEXT or BYTE strings 2-80
on VARCHAR 2-80
use in expression 2-79
Less than or equal to (<=) relational operator 2-32
Less than (<) relational operator 2-31
LET statement
assigning values 8-25
executing a procedure 8-12
LIKE keyword
description of 2-37
using in WHERE clause 2-29
Local variables in SPL 8-20
Locale Intro-4, 1-17
Lock mode, TABLE 7-9
LOCK TABLE statement, locking a table explicitly 7-9
Locking
and concurrency 4-34
and integrity 7-3
deadlock 7-20
description of 7-6
granularity 7-7
lock duration 7-11
lock mode 7-19
not wait 7-19
wait 7-19
locks released at end of transaction 7-21
scope of lock 7-7
setting lock mode 7-19
specifying number of rows to lock 7-15
types of locks
database lock 7-8
exclusive lock 7-7
key lock 7-10
page lock 7-10
promotable lock 7-7, 7-11
row lock 7-10
shared lock 7-7
table lock 7-8
with
DELETE 6-4
update cursor 7-11
Logical log
and backups 4-33
description of 4-30
Logical operator
AND 2-36
NOT 2-36
OR 2-36
Logslice, description of 4-31
Loop
creating and exiting in SPL 8-27
exiting with RAISE exception 8-35
LOWER function, as string manipulation function 2-72
LPAD function, as string manipulation function 2-77

M

Machine notes Intro-14
MATCHES keyword
using GLS 2-42
using in WHERE clause 2-29
MATCHES relational operator
how locale affects 2-42
in WHERE clause 2-37
MAX function, as aggregate function 2-60
MDY function, as time function 2-63
Message file
error messages Intro-13
MIN function, as aggregate function 2-60
MODE ANSI keywords, specifying transactions 4-32
MONEY data type
in INSERT statement 4-8
international money formats 1-17
MONTH function, as time function 2-63
Multiple-table join 2-95
Multithreaded application, definition of 5-5

N

Naming convention, tables 2-101
Natural join 2-93
NCHAR data type, querying on 2-11
Nested ordering, in SELECT 2-15
Nonlogging tables Intro-7, 4-31
Not equal (!=) relational operator 2-31
NOT logical operator 2-36
NOT relational operator 2-32
Null value
detecting in ESQL 5-16
in INSERT statement 4-8
testing for 2-35
with logical operator 2-36
NVARCHAR data type, querying on 2-11
NVL function 2-85

O

Object mode, description of 4-26
ON EXCEPTION statement
scope of control 8-33
trapping errors 8-32
user-generated errors 8-34
On-line manuals Intro-12
onload utility 4-34
onunload utility 4-34
OPEN statement
activating a cursor 5-21
opening select or update cursors 5-21

Opening a cursor 5-21, 5-24
OR logical operator 2-36
OR relational operator 2-33
ORDER BY keywords
and GLS 2-25
ascending order 2-14
DESC keyword 2-14, 2-25
display label with 2-56
multiple columns 2-15
relation to GROUP BY 3-6
restrictions in INSERT 4-11
restrictions with FOR UPDATE 6-8
select columns by number 2-24
sorting rows 2-13
Outer join Intro-7, 3-20
Output from TRACE statement 9-14
Owner-privileged procedure 8-17

P

Page lock 7-10
Parameter, to a stored procedure 8-29
Parts explosion 5-27
Performance
depends on concurrency 7-3
increasing with stored procedures 8-4
Platform icons Intro-10
PREPARE statement
description of 5-30
error return in SQLERRD 5-10
multiple SQL statements 5-31
Primary key
definition of 4-21
in fragmented table 3-15
Primary key constraint, definition of 4-23
Printed manuals Intro-13
Privilege
database-level 4-17
DBA-privileged procedures 8-17
default for stored procedures 8-18
displaying 4-19
Execute 8-18
needed
to modify data 4-17
on stored procedures 8-17
overview 1-9
owner-privileged procedures 8-17
revoking on stored procedure 8-20
Product icons Intro-10
Program group
Documentation notes Intro-14
Release notes Intro-14
Projection, definition of 2-7
Project, description of 1-14
Promotable lock 7-7, 7-11
PUT statement
constant data with 6-12
count of rows inserted 6-11
sends returned data to buffer 6-10

Q

Qualifier, existential 3-36
Query
audit 3-39
compound 3-40
cyclic 4-26
self-referencing 4-26
stated in terms of data model 1-7

R

RAISE EXCEPTION statement, exiting a loop 8-27
RANGE function, as aggregate function 2-60
Read Committed isolation level (ANSI) 7-14
Read Uncommitted isolation level (ANSI) 7-14
Recursion, in a stored procedure 8-29
Recursive relationship, example of 5-27
Reentrant trigger, description of 9-13
REFERENCING clause 9-9
Referential constraint, definition of 4-23
Referential integrity, definition of 4-22
Related reading Intro-15
Relational database, definition of 1-11
Relational model
description of 1-11
join 2-8
projection 2-6
selection 2-6
Relational operation 2-5
Relational operator
BETWEEN 2-32
EXISTS 3-31
IN 3-31
in a WHERE clause 2-29 to 2-31
LIKE 2-37
NOT 2-32
NULL 2-35
OR 2-33
Release notes
location of Intro-14
program item Intro-14
Repeatable Read isolation level (Informix and ANSI) 7-17
REPLACE function, as string manipulation function 2-74
Replication
of data 4-35
transparency 4-36
RETURN statement, exiting a loop 8-27
REVOKE statement, in embedded SQL 5-34 to 5-36
ROLLBACK WORK statement
closes cursors 7-21
releases locks 7-11, 7-21
setting SQLCODE 6-5
Row
definition of 1-13, 2-5
deleting 4-4
finding number of rows processed 2-82
finding rows that a user modified 3-18
in relational model 1-13

inserting 4-7
specifying number of rows returned with FIRST clause 2-46
Row lock 7-10
Rowid
using in self-join 3-15
using to find modified rows 3-18
using to locate internal row numbers 3-16
RPAD function, as string manipulation function 2-78

S

Sample-code conventions Intro-11
Scroll cursor
active set 5-25
definition of 5-23
SCROLL keyword, using in DECLARE 5-24
Select cursor
opening 5-21
using 5-21
Select list
display label 2-52
expressions in 2-49
functions in 2-57 to 2-82
labels in 3-44
selecting all columns 2-12
selecting specific columns 2-18
specifying a substring in 2-27
SELECT statement
active set 2-28
advanced 3-4 to 3-52
aggregate functions in 2-57, 2-68
alias names 2-96
and end-of-data return code 6-14
assigning values with 8-25
compound query 3-40
cursor for 5-20
date-oriented functions in 2-63
display label 2-52
DISTINCT keyword 2-20
embedded 5-14 to 5-17
FIRST clause 2-46
for joined tables 2-100
for single tables 2-11 to 2-82
functions 2-57 to 2-82
GROUP BY clause 3-4
HAVING clause 3-8
INTO clause with ESQL 5-14
INTO TEMP clause 2-100
join 2-90 to 2-96
multiple-table 2-88
natural join 2-93
ORDER BY clause 2-13
outer join 3-20 to 3-27
rowid 3-20
SELECT clause 2-12 to 2-28
selecting a substring 2-27
selecting expressions 2-49
selection list 2-12
self-join 3-11
simple 2-3 to 2-100
single-table 2-11
singleton 2-28
subquery 3-30 to 3-39
UNION operator 3-40
using
for join 2-8
for projection 2-7
for selection 2-6
Selection, description of 2-6
Select, description of 1-14
Self-join
assigning column names with INTO TEMP 3-12
description of 3-11
Self-referencing query 3-11, 4-26
Semantic integrity 4-21
Sequential cursor, definition of 5-23
SERIAL data type
finding last SERIAL value inserted 2-82
generated number in SQLERRD 5-10
inserting a starting value 4-8
Serializable isolation level (ANSI), description of 7-17
Session ID, returned by DBINFO function 2-83
SET clause 4-15
Set difference 3-50
Set intersection 3-48
SET ISOLATION statement
compared with SET TRANSACTION statement 7-12
description of 7-12
SET keyword, use in UPDATE 4-13
SET LOCK MODE statement, description of 7-19
Set operation 3-39
difference 3-50
intersection 3-48
union 3-40
SET TRANSACTION statement, compared with SET ISOLATION statement 7-13
Singleton SELECT statement 2-28
SITENAME function, in SELECT statement 2-81, 3-19
Software dependencies Intro-4
SOME keyword, beginning a subquery 3-31
Sorting
as affected by a locale 2-25
effects of GLS 2-25
nested 2-15
with ORDER BY 2-14
SPL
definition of 8-3
flow control statements 8-26
global variable in 8-20
local variable in 8-20
program variable 5-6
relation to SQL 8-3
SQL
Application Programming Interfaces 5-4
compliance of statements with ANSI standard 1-16
cursor 5-20
description of 1-15
error handling 5-17
history 1-15
Informix SQL and ANSI SQL 1-16
interactive use 1-18
standardization 1-15
SQL Communications Area (SQLCA)
altered by end of transaction 6-5

description of 5-8
inserting rows 6-11
SQLCODE field
after opening cursor 5-21
and FLUSH operation 6-11
description of 5-9
end of data on SELECT only 6-14
end of data signalled 5-17
set by DELETE statement 6-4
set by PUT statement 6-11
SQLERRD array
count of deleted rows 6-4
count of inserted rows 6-11
count of rows 6-14
description of 5-10
syntax of naming 5-8
SQLERRM character array 5-13
SQLSTATE variable
in non-ANSI-compliant databases 5-18
using with a cursor 5-21
SQLWARN array
description of 5-11
syntax of naming 5-8
with PREPARE 5-31
Standard deviation, aggregate function 2-61
Static SQL 5-5
STDEV function, as aggregate function 2-61
Stored procedure
altering 8-16
as triggered action 9-11
branching 8-26
comments in 8-8
creating from an embedded language 8-7
creating from DB-Access 8-6
DBA-privileged, using with triggers 8-17
debugging 8-14
default privileges 8-18
DEFINE statement 8-21
definition of 8-4
displaying contents 8-11
displaying documentation 8-11
executing 8-12
general programming 1-19
granting privileges on 8-18, 8-19
header 8-30
in SELECT statements 2-86
introduction to 8-3
looping 8-27
name confusion with SQL functions 8-25
nesting and privileges 8-19
owner-privileged 8-17
privileges necessary at execution 8-18
program flow control 8-26
recursion 8-29
REFERENCES clause 8-22
returning values 8-30
revoking privileges on 8-20
SPL expressions 8-25
tracing triggered actions 9-13
use 8-4
variable 8-20
variables, scope of 8-22
Stored procedure language. *See* SPL.
stores7 database Intro-5
Subquery
correlated 3-30, 4-26
in DELETE statement 4-6
in SELECT statement 3-30 to 3-39
in UPDATE statement
with SET clause 4-14
with WHERE clause 4-13
Subscripting
in a WHERE clause 2-44
SPL variables 8-22
Subservient table 3-20
SUBSTR function, as string manipulation function 2-76
Substring 2-27, 8-22
SUBSTRING function, as string manipulation function 2-75
SUM function, as aggregate function 2-60
System catalog
privileges in 4-19
querying 4-19
sysprocbody 8-11
systabauth 4-19
System descriptor area, description of 5-32

T

Table
accessing 2-101
accessing a table not in the current database 2-38
description of 1-12
in relational model 1-12
loading data
with external tables 4-34
with onload utility 4-34
lock 7-8
logging 4-31
names 2-101
nonlogging 4-31
TABLE lock mode Intro-7, 7-9
Temporary table
and active set of cursor 5-25
assigning column names 3-12
example 4-12
TEXT data type
restrictions
with GROUP BY 3-6
with LIKE or MATCHES 2-37
with relational expression 2-29
using LENGTH function on 2-80
TEXT value, displaying 2-11
Time function, description of 2-63
TODAY function, in constant expression 2-81, 4-9
TO_CHAR function, as conversion function 2-69
TO_DATE function, as conversion function 2-70
TRACE statement
debugging a stored procedure 8-14
output from 9-14
Transaction
cursors closed at end 7-21
description of 4-28
example with DELETE 6-5
locks held to end of 7-11
locks released at end of 7-11, 7-21
transaction log 4-30
use signalled in SQLWARN 5-11
Transaction logging
contents of log 4-33
description of 4-30

Trigger
  creating 9-4
  definition of 9-3
  disabled mode 4-28
  enabled mode 4-28
  name assigning 9-5
  reentrant, description of 9-13
  when to use 9-3
Trigger event
  definition of 9-5
  example of 9-5
Triggered action
  BEFORE and AFTER 9-7
  FOR EACH ROW 9-9
  generating an error message 9-15
  in relation to triggering statement 9-6
  statements 9-3
  tracing 9-13
  using 9-7
  using stored procedures 9-11
  WHEN condition 9-10
Truncation, signalled in SQLWARN 5-12

U

UNION operator
  description of 3-40
  display labels with 3-44
Union set operation 3-40
UNIQUE keyword, in SELECT statement 2-20
Update cursor, definition of 6-15
UPDATE statement
  and end of data 6-14
  description of 4-12
  embedded 6-15 to 6-17
  multiple assignment 4-15
  number of rows 5-10
  preparing 5-31
  restrictions on subqueries 4-15
  using a join to update a column 4-17
UPPER function, as string manipulation function 2-72
USER function, in expression 2-80, 3-18
USING keyword, in EXECUTE statement 5-31
Utility program
  onload 4-34
  onunload 4-34

V

VALUES clause, in INSERT statement 4-7
VARCHAR data type, using LENGTH function on 2-80
VARCHAR value, displaying 2-11
Variable
  global, in SPL 8-20, 8-21
  in SPL 8-20
  scope in SPL 8-22
  with same name as a keyword 8-23
VARIANCE function, as aggregate function 2-62
Version number, returned by DBINFO function 2-83
Violations table, using with filtering mode 4-27

W

Warnings, with stored procedures at compile time 8-10
WEEKDAY function, as time function 2-63, 2-66
WHERE clause
  Boolean expression in 2-36
  comparison condition 2-29
  date-oriented functions in 2-66
  host variables in 5-15
  in DELETE 4-4 to 4-7
  relational operators 2-29
  selecting rows 2-28
  subqueries in 3-31
  subscripting 2-44
  testing a subscript 2-44
  wildcard comparisons 2-37
  with NOT keyword 2-32
  with OR keyword 2-33
WHERE CURRENT OF clause
  in DELETE statement 6-7
  in UPDATE statement 6-15
WHERE keyword
  null data tests 2-35
  range of values 2-32
WHILE statement, looping in a stored procedure 8-27
Wildcard character, asterisk 2-12
Wildcard comparison in WHERE clause 2-37 to 2-44
WITH HOLD keywords, declaring a hold cursor 7-23
WITH LISTING IN keywords, warnings in a stored procedure 8-10

X

X/Open compliance
  level Intro-15

Y

YEAR function, as time function 2-63

Symbols

!=, not equal, relational operator 2-31
<, less than, relational operator 2-31
<=, less than or equal to, relational operator 2-32
=, equals, relational operator 2-30, 2-90
>, greater than, relational operator 2-31
>=, greater than or equal to, relational operator 2-32
?, question mark
  as placeholder in PREPARE 5-30