IBM Informix Guide to SQL

Tutorial

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

Preface

This book is a tutorial on the Structured Query Language (SQL) as it is imple-mented by Informix products. *The Informix Guide to SQL: Tutorial* and its com-panion volume *The Informix Guide to SQL: Reference*, tell you how to create, manage, and use relational databases with Informix software tools.

The Informix Guide to SQL: Tutorial was written for people who already know how to use computers. To use this book effectively, you regularly should use a computer—either a desktop workstation or a terminal connected to a larger machine—in the course of your daily work. You need to know how to start programs, create and copy files, and execute other common commands of your computer operating system.

You also need to have the following Informix software:

- An **INFORMIX-OnLine** *database server* or an **INFORMIX-SE** database server

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

- An Informix application development tool, such as **INFORMIX-SQL**, **INFORMIX-4GL**, or **DB-Access**

   The application development tool enables you to compose queries, send them to the database server, and view the results that the database server returns. You can use **DB-Access** to try out all the SQL statements described in this guide.

Summary of Chapters

The Informix Guide to SQL: Tutorial includes the following chapters:

- This Preface provides general information about the book and lists additional reference materials that can help you understand relational database management.

- The Introduction tells how SQL fits into the Informix family of products and books, explains how to use this book, introduces the demonstration database from which the product examples are drawn, describes the **Informix Messages and Corrections** product, and lists the new features for Version 5.0 of Informix server products.

- Chapter 1, "Database Fundamentals," contains an overview of database terminology and defines some important terms and ideas that are used throughout the book.

- Chapter 2, "Simple SELECT Statements," begins the exploration of making simple queries to fetch and display database data.

- Chapter 3, "Advanced SELECT Statements," discusses making advanced queries to fetch and display database data.

- Chapter 4, "Optimizing Your Queries," defines techniques to refine and optimize your queries, introduces and explains the query Optimizer, and discusses the time costs of various operations.

- Chapter 5, "Statements That Modify Data," describes the statements you use to insert, delete, or update data, and introduces the concepts of database privileges, maintaining data integrity, and archiving data.

- Chapter 6, "SQL in Programs," discusses calling the database server, retrieving rows, and embedding data.

- Chapter 7, "Programs That Modify Data," provides an in-depth look at INSERT, DELETE, and UPDATE statements, as well as a complete discussion on concurrency and locking data.

- Chapter 8, "Building a Data Model," describes the components of a data model and provides a step-by-step procedure for building one.

- Chapter 9, "Implementing the Model," tells you how to define the database domains and create the database.

- Chapter 10, "Tuning the Model," discusses many of the details that help you set up an efficient database model, including disk storage, calculating table sizes, managing indexes, and maximizing concurrency.

- Chapter 11, "Security and Views," details how you can ensure data security by granting privileges and using stored procedures and views.
- Chapter 12, "Networks and Distribution," discusses networks and how you can best set up a database to work over one.

Related Reading

If you have no prior experience with database management, you may want to refer to an introductory text such as C. J. Date's *Database: A Primer* (Addison-Wesley Publishing, 1983). If you want more technical information on database management, consider consulting the following texts, also by C. J. Date:

- *An Introduction to Database Systems, Volume I* (Addison-Wesley Publishing, 1990)
- *An Introduction to Database Systems, Volume II* (Addison-Wesley Publishing, 1983)

This guide assumes you are familiar with your computer operating system. If you have limited UNIX system experience, you may want to look at your operating system manual or a good introductory text before you read this tutorial.

The following are some suggested texts about UNIX systems:

- *A Practical Guide to the UNIX System*, Second Edition, by M. Sobell (Benjamin/Cummings Publishing, 1989)
- *A Practical Guide to UNIX System V* by M. Sobell (Benjamin/Cummings Publishing, 1985)
- *UNIX for People* by Birns, Brown, and Muster (Prentice-Hall, 1985)

Table of Contents

Chapter 4   **Optimizing Your Queries**

Chapter 5   **Statements That Modify Data**

Chapter 6   **SQL in Programs**

Chapter 11   **Security, Stored Procedures, and Views**

Introduction

The Structured Query Language, or SQL as it is commonly known, is an English-like language that you can use when working with databases. The SQL provided with Informix products is an enhanced version of the industry-standard query language developed by IBM. SQL allows you to create and use databases.

Informix Products That Use SQL

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

Informix UNIX products work with either an **INFORMIX-OnLine** database server or an **INFORMIX-SE** database server. If you are running applications on a network, you can use an Informix client/server product such as **INFOR-MIX-NET** or **INFORMIX-STAR**. **INFORMIX-NET** is the communications facility for multiple **INFORMIX-SE** database servers. **INFORMIX-STAR** allows distributed database access to multiple **INFORMIX-OnLine** database servers.

Other Useful Documentation

You may want to refer to a number of related Informix documents that complement the *The Informix Guide to SQL: Tutorial.*

- A companion volume to the Tutorial, *The Informix Guide to SQL: Reference,* provides full information on the structure and contents of the demonstration database that is provided with all Informix application development tools. It includes details of the Informix system catalog, describes Informix and common UNIX environment variables that should be set, and defines column data types supported by Informix products. Further, it provides a detailed description of all the SQL statements supported by Informix products. It also contains a glossary of useful terms.

- You, or whoever installs your Informix products, should refer to the *UNIX Products Installation Guide* for your particular release to ensure that your Informix product is properly set up before you begin to work with it.

- If you are using your Informix product across a network, you may also want to refer to the appropriate *INFORMIX-NET/INFORMIX-STAR Installation and Configuration Guide.*

- Depending on the database server you are using, you or your system administrator need either the *INFORMIX-OnLine Administrator's Guide* or the *INFORMIX-SE Administrator's Guide.*

- When errors occur, you can look them up, by number, and find their cause and solution in the *Informix Error Messages* manual. If you prefer, you can look up the error messages in the on-line message file described in the section "ASCII and PostScript Error Message Files" later in this Introduction.

How to Use This Manual

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

Typographical Conventions

The Informix Guide to SQL: Tutorial uses a standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth. The following typographical conventions are used throughout the manual:

italics	When new terms are introduced, they are printed in italics.
boldface	Database names, table names, column names, filenames, utilities, and other similar terms are printed in boldface.
computer	Information that your Informix product displays and information that you enter are printed in a computer typeface.
KEYWORD	All keywords appear in uppercase letters.

Additionally, when you are instructed to "enter" or "execute" text, immediately press RETURN after the entry. When you are instructed to "type" the text, no RETURN is required.

Example Code Conventions

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

For example, you might see the following example code:

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

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

Also note that dots in the example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

Useful On-Line Files

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

Documentation Notes	describe feature and performance topics not covered in the manual or which have been modified since publication. The file containing the Documentation Notes for this manual is called **SQLTDOC_5.0**.
Release Notes	describe feature differences from earlier versions of Informix products and how these differences may affect current products. The file containing the Release Notes for Version 5.0 of Informix products is called **ENGREL_5.0**.
Machine Notes	describe any special actions required to configure and use Informix products on your machine.

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

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

ASCII and PostScript Error Message Files

Informix software products provide ASCII files that contain all the Informix error messages and their corrective actions. To access the error messages in the ASCII file, Informix provides scripts that let you display error messages on the terminal or print formatted error messages.

The optional **Informix Messages and Corrections** product provides Post-Script files that contain the error messages and their corrective actions. If you have installed this product, you can print the PostScript files on a PostScript printer.

Using the ASCII Error Message File

You can use the file that contains the ASCII text version of the error messages and their corrective actions in two ways:

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

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

   $INFORMIXDIR/msg/errmsg.txt

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

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

The *finderr* Script

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

```
finderr ──────────────┐  ┌──── msg_num ────┐──────────────────────┤
                      └─────────────────┘
              ┌──── - ────┐
              └──── + ────┘
```
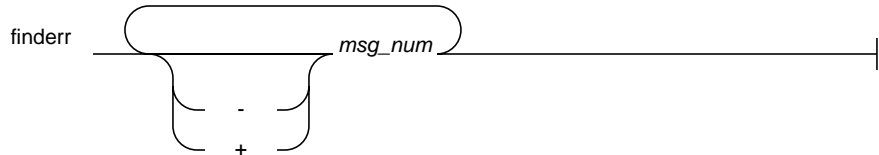
msg_num is the number of the error message to display.

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

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

```
finderr -359
```

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

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

The *rofferr* Script

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

The **rofferr** script has the following syntax:
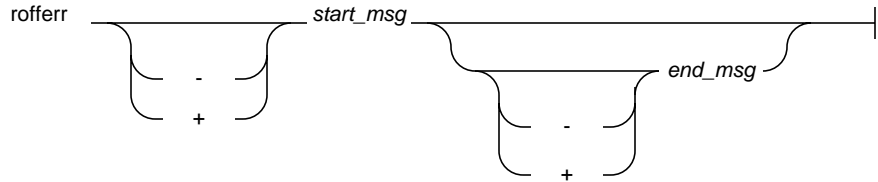
start_msg   is the number of the first error message to format. This error
message number is required.

end_msg   is the number of the last error message to format. This error
message number is optional. If you omit *end_msg*, only
start_msg is formatted.

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

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

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

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

Using the PostScript Error Message Files

Use the **Informix Messages and Corrections** product to print the error mes-
sages, and their corrective actions, on a PostScript printer. The PostScript
error messages are distributed in a number of files of the format **errmsg1.ps**,
errmsg2.ps, and so on. These files are located in the **$INFORMIXDIR/msg**
directory. Each file contains approximately 50 printed pages of error
messages.

The Demonstration Database

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

Most of the examples in this manual are based on the **stores5** demonstration database. The **stores5** database is described in detail and its contents are listed in Chapter 1 of *The Informix Guide to SQL: Reference.*

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

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

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

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

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

Creating the Demonstration Database on INFORMIX-OnLine

Use the following steps to create and populate the demonstration database in the **INFORMIX-OnLine** environment:

1. Set the INFORMIXDIR environment so that it contains the name of the directory in which your Informix products are installed. Set SQLEXEC to **$INFORMIXDIR/lib/sqlturbo**. (For a full description of environment variables, see Chapter 4 of *The Informix Guide to SQL: Reference.*)

2. Create a new directory for the SQL command files. Create the directory by entering

   ```
   mkdir dirname
   ```

3. Make the new directory the current directory by entering

   ```
   cd dirname
   ```

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

   ```
   dbaccessdemo5 dbname
   ```

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

To give someone else the SQL privileges to access the data, use the GRANT and REVOKE statements. The GRANT and REVOKE statements are described in Chapter 7 of *The Informix Guide to SQL: Reference.*

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

Creating the Demonstration Database on INFORMIX-SE

Use the following steps to create and populate the demonstration database in the **INFORMIX-SE** environment:

1. Set the INFORMIXDIR environment so that it contains the name of the directory in which your Informix products are installed. Set SQLEXEC to

$INFORMIXDIR/lib/sqlexec. (For a full description of environment variables, see Chapter 4 of *The Informix Guide to SQL: Reference.*)

2. Create a new directory for the demonstration database. This directory will contain the example command files included with the demonstration database. Create the directory by entering

   ```
   mkdir dirname
   ```

3. Make the new directory the current directory by entering

   ```
   cd dirname
   ```

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

   ```
   dbaccessdemo5 dbname
   ```

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

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

To give someone else access to the database that you created, grant them the appropriate privileges using the GRANT statement in **DB**-**Access**. To remove privileges, use the REVOKE statement. The GRANT and REVOKE statements are described in Chapter 7 of *The Informix Guide to SQL: Reference.*

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are compliant with ANSI Level 2 (published as ANSI X3.135-1989) on the **INFORMIX**-**OnLine** database server. They are compliant with ANSI Level 2 on the **INFORMIX**-**SE** database server with the following exceptions:

• Effective checking of constraints

• Serializable transactions

INFORMIX-**TP/XA** conforms to the *X/Open Preliminary Specification (April 1990), Distributed Transaction Processing: The XA Interface.*

New Features in Informix Server Products, Version 5.0

This section highlights the major new features implemented in Version 5.0 of Informix server products:

- Referential and Entity Integrity

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

- Stored Procedures

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

- Dynamic SQL

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

- Optimizer Enhancement

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

- Relay Module (**INFORMIX**-**NET** only)

  The new Relay Module component of **INFORMIX**-**NET** resides on the client machine in a distributed data processing environment and *relays* messages between the application development tool and an **INFOR-MIX**-**OnLine** or **INFORMIX**-**SE** database server through a network inter-

face. The Relay Module allows version 5.0 application development tools to connect to a remote database server without the need to run an Informix database server process on the client.

- Fast Indexing (**INFORMIX-OnLine** only)

  The version 5.0 **INFORMIX-OnLine** database server uses a new method of creating large indexes when you execute the CREATE INDEX statement. In this method, index entries are sorted prior to their insertion into the B+ tree structure, resulting in faster index creation.

- Two-Phase Commit (**INFORMIX-STAR** only)

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

- Support for Optical Media (**INFORMIX-OnLine/Optical** only)

  A new product, **INFORMIX-OnLine/Optical** , allows you to store and retrieve blob data using an optical storage subsystem. The new SQL statements and functions that support optical storage are described in the *INFORMIX-OnLine/Optical User Manual.*

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

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

Database Fundamentals

Chapter Overview

This is a book about databases, and about how you can exploit them using Informix software. Your real use of a database begins in Chapter 2 with the SELECT statement. 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, with emphasis on the following topics:

- What makes a database different from any collection of files?
- 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?

Databases: What and Why?

A *database* is *a collection of information*—but so is a simple computer file. What makes a database so special? There must be reasons; otherwise people would not spend so much money, effort, and computer time on databases. There are two fundamental differences. First, a database comprises not only data but a plan, or *model* of the data. Second, a database can be a common resource, used concurrently by many people.

The Data Model

The principal difference between information collected in a database and information in a file is in the way the data is organized. A 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 assigned to it by the data model. It may 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 things—prices, products, items, orders, and customers—are also roles specified by the data model. (See Figure 1-1.)

The data model is designed when the database is created; then units of data are inserted according to the plan laid out in the model. In some books, the term *schema* is used instead of *data model.*
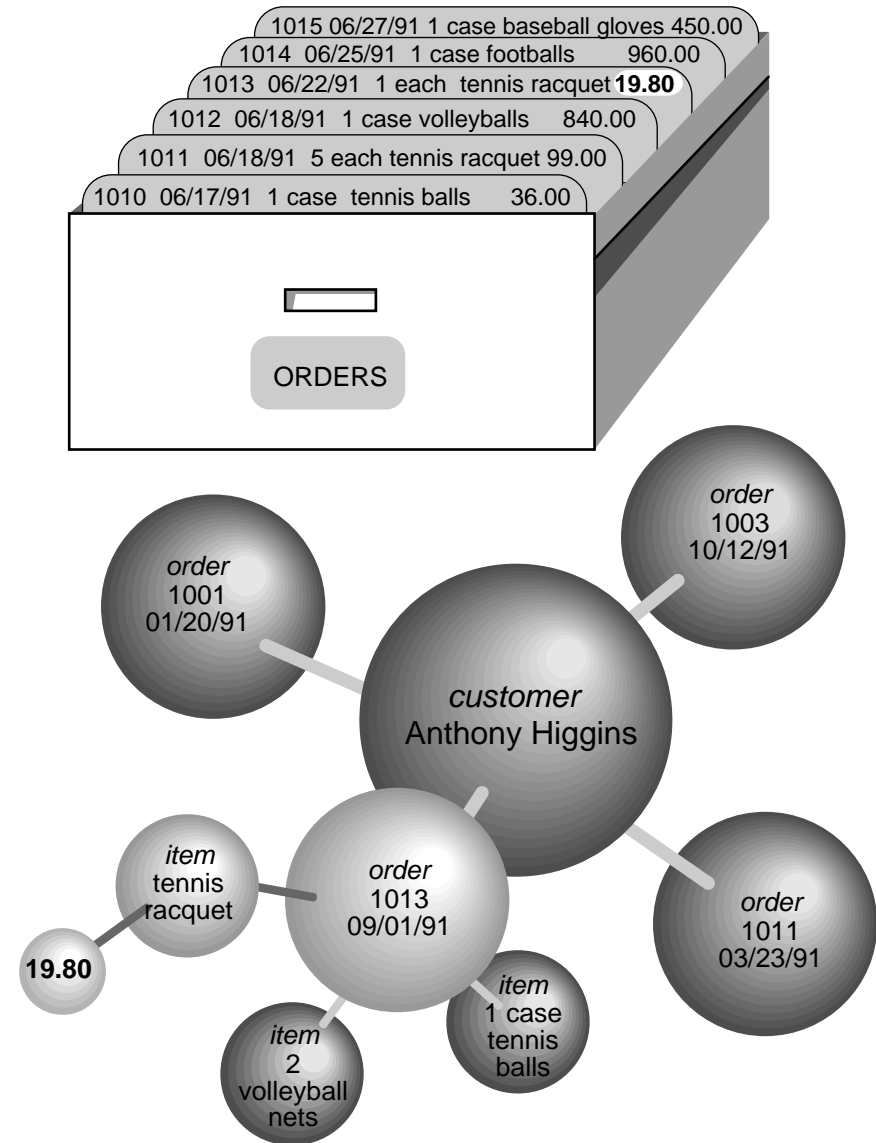
Figure 1-1   Illustration of a data model

Storing Data

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

A file may 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 as stored by a word processing program may contain very detailed structures describing the format of the document. However, only the word processing program itself 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 their data types as well, 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 Chapters 6 and 7 in this manual.

Querying Data

Yet another difference between a database and a file is in the way you can interrogate them. You can search a file sequentially, looking for particular values at particular physical locations in each record. That is, you might ask of a file, "What records have numbers under 20 in the fifth field of each record?" Figure 1-2 illustrates this type of search.
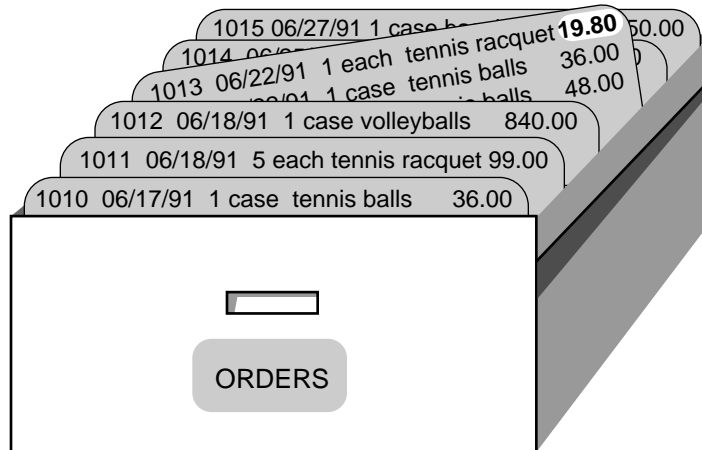
Figure 1-2 **Searching a file sequentially**

In contrast, when you query a database you use the terms defined by its model. To a database, you can pose 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 second quarter?" Figure 1-3 illustrates this type of query.

In other words, when you interrogate data 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 itself reflects the real world.

In this manual, Chapters 2 and 3 discuss the language you use for making queries. Chapters 8 through 11 discuss designing an accurate, robust data model for other people to query.
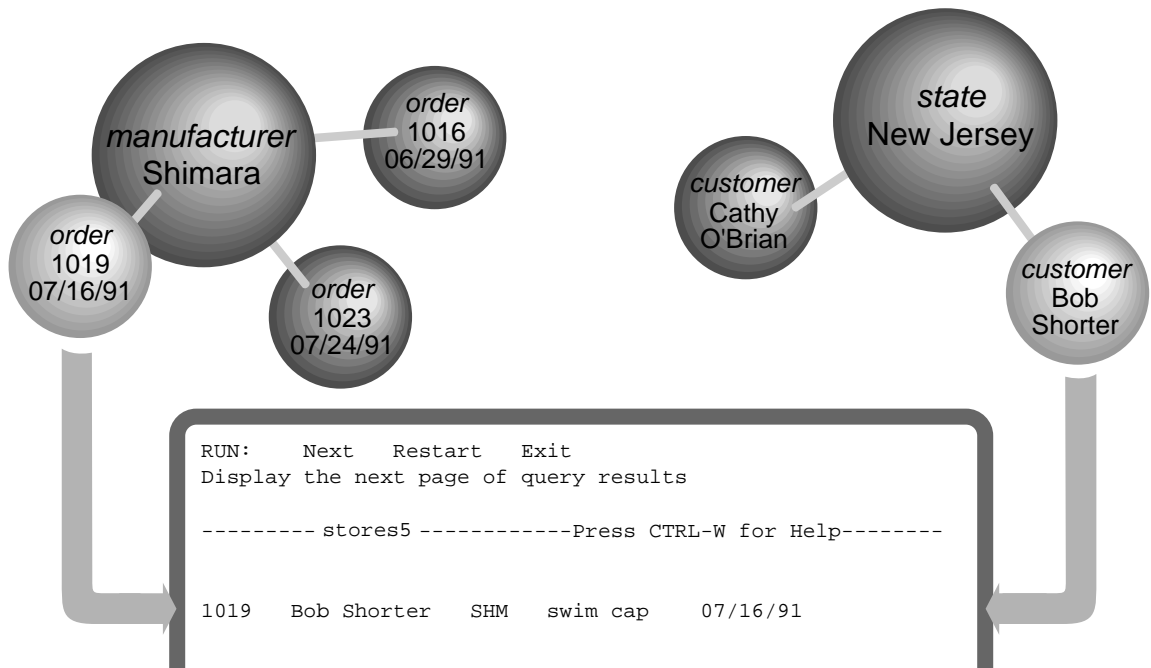
Figure 1-3   Querying a database

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 commands such as "Find every *stock item* with a *manufacturer* of Presta or Schraeder and increase its *price* by 13%." 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 5 of this manual.

Concurrent Use

A database can be a common resource for many computer 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 only among a select group of persons. Still other databases provide public access.

Informix database software provides the means to control database use. (See Figure 1-4.) When you design a database, you can perform any of these 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.

The facilities that make these and other things possible are discussed in Chapter 11 of this manual.

Query TABLE A and
TABLE B only

Update TABLE C Only

TABLE C

TABLE A

TABLE B

TABLE B,C

Modify and Update
TABLES A and B only

View Column 3, TABLE B and
Columns 1 and 2, TABLE C only
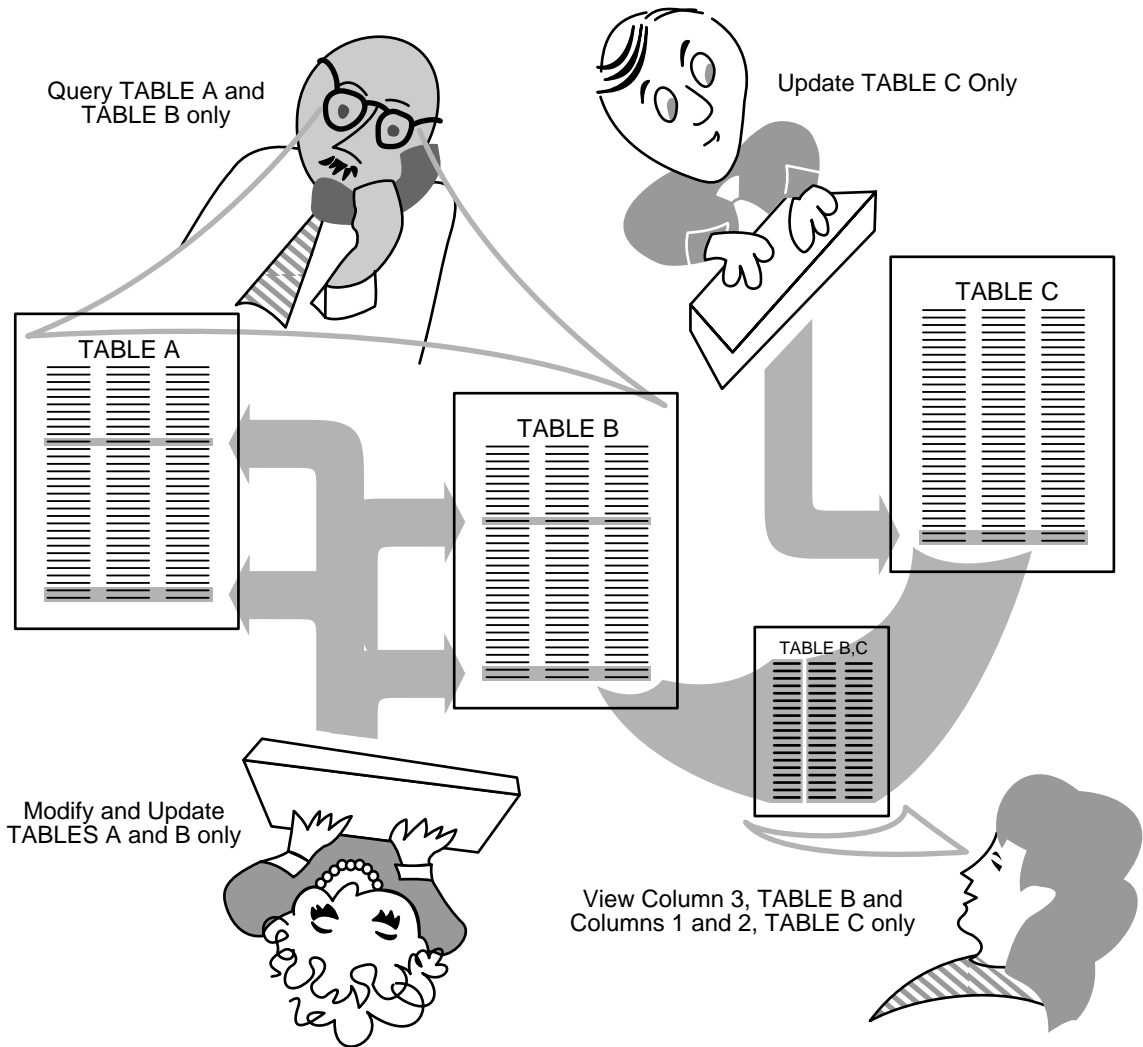
Figure 1-4   Sharing the database

Centralized Management

Databases that are used by many people are highly valuable and must be protected as important business assets. This creates two important problems: archiving and maintenance. The **INFORMIX-OnLine** database server allows these tasks to be centralized.

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 recreating the lost data, but the loss of productive time by the database users, and the loss of business and good will while the users are unable to work. A plan for regular archiving of critical databases can help avoid or mitigate these potential disasters.

A large database used by many people 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 and correct them. If rapid response is important, someone must study and analyze the performance of the system and find the causes of slow responses.

Group and Private Databases

Some Informix database servers are designed to manage relatively small databases that are used privately by individuals, or that are shared among a small group of users.

These database servers (for example, **INFORMIX-SE** for the UNIX operating system) store databases in files managed by the host operating system. These databases can be archived using the same procedures for backing up files that work with other computer files; that is, copying the files to another medium when they are not in use. The only difference is that when a database is archived its transaction log file must be reset to empty. (The use of transaction logs is discussed in Chapter 7; Chapter 10 has more information on archiving.)

Performance problems that arise in group and private databases are usually related to particular queries that take too long. Chapter 4 deals in depth with the reasons why a query takes more or less time. After you understand all the features of the SELECT statement and the alternative ways of stating a query, as covered in Chapters 2 and 3, you can use the tips in Chapter 4 to improve the performance of the queries that otherwise might take excessive amounts of time.

Essential Databases

The **INFORMIX**-**OnLine** database server is designed to manage large databases with requirements for high reliability, high availability, and high performance. While it supports private and group databases very well, it is at its best managing the databases that are essential if your organization is to carry out its work successfully.

INFORMIX-**OnLine** gives its operator the ability to make archival copies while the databases are in use. It also allows incremental archiving (archiving only modified data), an important feature when a full copy could take many reels of tape.

INFORMIX-**OnLine** has an interactive monitor program by which its operator (or any user) can monitor the activities within the database server to see when bottlenecks are developing. It also comes with utility programs to analyze its use of disk storage.

Chapter 10 contains an overview of the **INFORMIX**-**OnLine** disk storage methods, as part of the necessary background information to the main topic. However, all the details of using and managing **INFORMIX**-**OnLine** are contained in the *INFORMIX-OnLine Administrator's Guide.*

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.

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* comprising *rows* and *columns*.

Tables

A database is a collection of information 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 table from the demonstration database is shown in Figure 1-5.

stock Table					
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	box	12/box

Figure 1-5 **The** stock *table from the demonstration database distributed with all Informix products*

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

A database is primarily 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 and not others.

Columns

Each column of a table stands for one *attribute*, that 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; that is, one particular, individual example of that entity. Each row of the **stock** table stands for one item of merchandise that is sold by the sporting-goods store.

Tables, Rows, and Columns

You now understand that the relational model of data is a very simple way of organizing data to reflect the world using these simple corresponding relationships:

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.

There are some rules about how you choose entities and attributes, but they are important only when you are designing a new database. (Database design is covered in Chapters 8 through 11 of this manual.) The data model in an existing database already is set. To use the database, you only need to know the names of the tables and columns and how they correspond to the real world.

Operations on Tables

Since a database is really a collection of tables, database operations are operations on tables. The relational model supports three fundamental operations, two of which are illustrated in Figure 1-6. (All three operations are defined in more detail, with many examples, in Chapters 2 and 3 of this manual.)

To *select* from a table is to choose certain rows, leaving others aside. One selection that could be made on the **stock** table is "select all rows in which the manufacturer code is HRO and the unit price is between 100.00 and 200.00."

To *project* from a table is to choose certain columns, leaving others aside. One projection that can be made from the **stock** table is "show me the **stock_num**, **unit_descr**, and **unit_price** columns only."

A table contains information about only one entity; when you want information about multiple entities, you must *join* their tables. There are many different ways of joining tables. (The join operation is the subject of Chapter 3 of this manual.)

stock Table

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	◄ SELECTION
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	

P   R   O   J   E   C   T   I   O   N

Figure 1-6   Illustration of selection and projection

Structured Query Language

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 second 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("7/1/91") AND DATE("7/30/91")
```

This question is a sample of the 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. SQL includes about 67 statements, from ALLOCATE DESCRIPTOR to WHENEVER.

All of the SQL statements are specified in detail in *The Informix Guide to SQL: Reference.* Most of the statements are used infrequently—while setting up or tuning a database. Most people spend their time using three or four statements at most.

One statement, SELECT, is in almost constant use. It is the only statement with which you can retrieve data from the database. It is also the most complicated of the statements, and all of the next two chapters of this book are devoted to exploring 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 controlling 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 of these SQL implementations differed in small ways from each other 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 is supported by Informix products is highly 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% of the SQL use for a typical person.

However, the extensions do exist, and that creates a conflict. Thousands of Informix customers have embedded Informix-style SQL in programs and stored queries. 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 resolved 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.

This resolution is fair but makes the SQL documentation more complicated. Wherever there is a difference between Informix and ANSI SQL, *The Informix Guide to SQL: Reference* describes both versions. Since you probably intend to use only one version, you will have to ignore the version you do not need.

ANSI-Compliant Databases

You can designate a database as ANSI-compliant by using the MODE ANSI keywords when you create it. Within such a database, certain characteristics of the ANSI standard apply. For example, all actions that modify data automatically take place within a transaction, 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 Informix Guide to SQL: Reference.*

The Database Software

You only can access your database with the help of two layers of sophisticated software. You work directly with the top layer, and it uses the bottom layer to access the data. You command both layers using the Structured Query Language.

The 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.

The Applications

A *database application*, or simply *application*, is a program that uses the database. It does so by calling on the database server. At its simplest, the application sends SQL commands 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 command to insert a row, and passes this command to the database server for execution.

There are several types of applications. Some allow you to access the database interactively using SQL; others present the stored data in different forms related to its use.

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 **INFORMIX**-**SQL** are two such programs. They assist you in composing SQL statements, then they pass your SQL to the database server for execution and display the results to you.

Alternatively, you may use a **Wingz** spreadsheet on a workstation or personal computer. **Wingz**, with its optional **DataLink** extension, can talk to a database server. Using the **Wingz** language, **HyperScript**, you can write SQL statements and have rows of data returned to your spreadsheet for display.

Reports and Forms

After you perfect a query to return precisely the data in which you are interested, you need to format the data for display as a report or form on the terminal screen. **ACE** is the report generator for **INFORMIX**-**SQL**. You provide to it a SELECT statement that returns the rows of data, and a report specification stating how the pages of the report are to be laid out. **ACE** compiles this information into a program that you can run whenever you want to produce that report. The **INFORMIX**-**QuickStep/Report** product provides another, less technical way to create a report program.

PERFORM is the module of **INFORMIX**-**SQL** that generates interactive screen forms. You prepare a form specification that relates display fields on the screen to columns within tables in the database. **PERFORM** compiles this specification into a program that you can run at any time. When run, the form program interrogates the database to display a row or rows of data on the

screen in the format you specified. You can arrange the form so that the user can type in sample values and have matching rows returned from the database (the *query-by-example* feature).

For more information on these products, refer to the manuals that come with them.

General Programming

Programs that you write can 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 in any way you choose. You also can write programs that take data from any source in any format, prepare it, and insert it into the database.

The most convenient programming language for this kind of work is **INFOR-MIX-4GL**, a nonprocedural language designed expressly for writing database applications. However, you can also communicate with an Informix database server from programs written in C, COBOL, Ada, and FORTRAN.

You also can write programs called stored procedures to work with database data and objects. The stored procedures that you write are stored directly in a database within tables. You can then execute a stored procedure from **DB-Access** or an embedded-language program.

Chapters 6 and 7 of this manual present an overview of how SQL is used in programs.

Applications and Database Servers

Every program that uses data from a database operates the same way. Regardless of whether it is a packaged program such as **INFORMIX-SQL**, a report program compiled by **ACE** or **QuickStep/Report**, or a program that you wrote using **INFORMIX-4GL** or embedded SQL, you find the same two layers in every case:

1. An application that interacts with the user, prepares and formats data, and sets up SQL statements.

2. A database server that manages the database and interprets the SQL.

All the applications converge on the database server, and only the database server manipulates the database files on disk. This concept is illustrated in Figure 1-7.
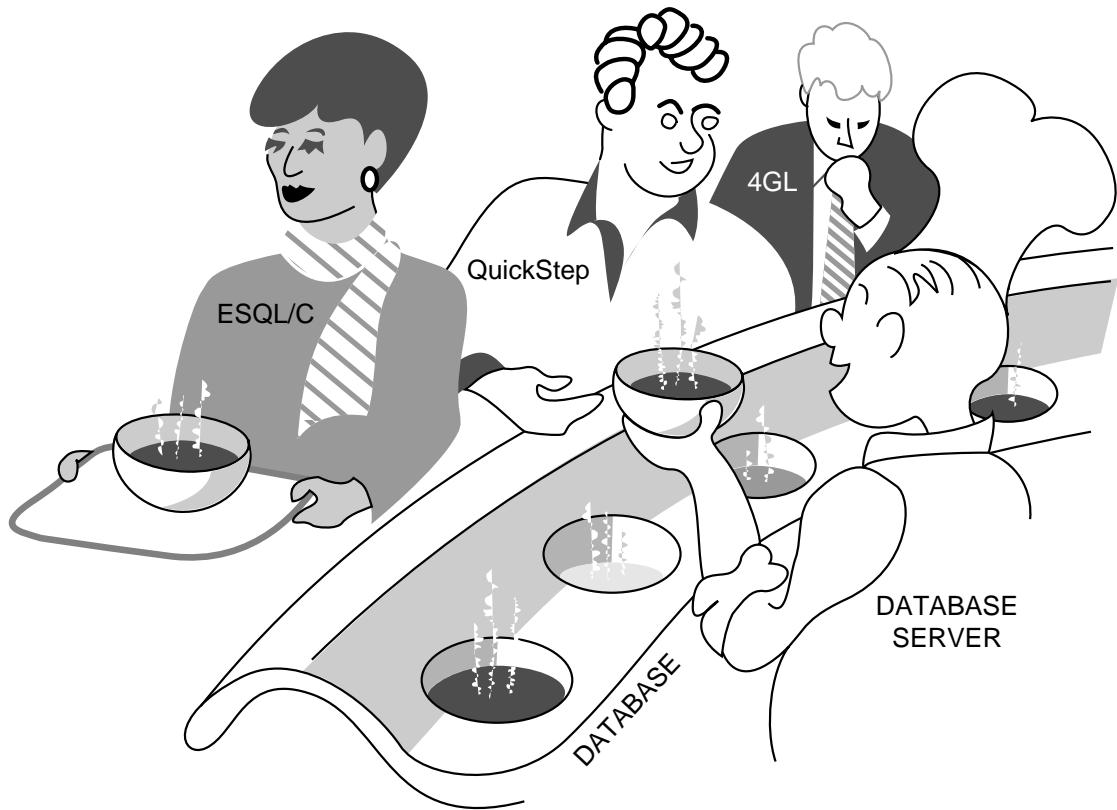
Figure 1-7   Many applications converging on one database server

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.

A database can be used, and even modified, by a number of computer users working concurrently. Different users can be given different views of the contents of a database, and their access to those contents can be restricted in several ways.

A database can be crucially important to the success of an organization and can require central administration and monitoring. The **INFORMIX**-**OnLine** database server caters to the needs of *essential* applications; both **INFOR-MIX**-**OnLine** and **INFORMIX**-**SE** support smaller databases for private or group use.

You control and query a database using the Structured Query Language, which was pioneered by IBM and standardized by ANSI. You will probably take advantage of the Informix extensions to the ANSI-defined language, but your Informix tools also makes 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, other vendors, or your colleagues.

Simple SELECT Statements

Chapter Overview

SELECT is the most important and the most complex SQL statement. It is used, 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 on 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.

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.

- 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, enable you to perform more complex data retrieval. They are introduced in Chapter 3. Another clause, INTO, is used to specify the program or host variable to receive data from a SELECT statement in the **INFORMIX-4GL** and **INFORMIX-ESQL** products. Complete syntax and rules for using the SELECT statement are shown in Chapter 7 of *The Informix Guide to SQL: Reference*.

Some Basic Concepts

The SELECT statement, unlike INSERT, UPDATE, and DELETE statements, does not modify the data in a database. It is used simply to query on the data. Whereas only one user at a time can modify data, any number of users can query on or select the data concurrently. The statements that modify data are discussed in Chapter 5. The INSERT, UPDATE, and DELETE statements appear in Chapter 7 of *The Informix Guide to SQL: Reference*.

Privileges

Before you can query on data, you must have CONNECT privilege to the database and SELECT privilege to the tables in it. These privileges normally are granted to all users as a matter of course. Database access privileges are discussed in "Chapter Overview" on page 11-3 of this manual and in the GRANT and REVOKE statements in Chapter 7 of *The Informix Guide to SQL: Reference*.

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 file that contains information on the database; or from a *view*, which is a virtual table created to contain a customized set of data. System catalog tables are shown in Chapter 2 of *The Informix Guide to SQL: Reference.* Views are discussed in Chapter 7 of *The Informix Guide to SQL: Reference.*

Relational Operations

A *relational operation* involves manipulating one or more tables or *relations* to result in another table. The three kinds of relational operation 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.

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

The result of this query 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 Interactive Editor screen, the data is
displayed vertically instead of horizontally.)

```
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      543 Nassau Street
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.

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

The result of this query contains the same number of rows as the **customer** table, but it *projects* only a subset of its columns.

```
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 returns some of the rows and some of the columns in a table.

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

The result of this query contains a subset of the rows and a subset of the columns in the **customer** table.

```
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. Figure 2-1 uses a subset of the **items** and **stock** tables to illustrate the concept of a join.

```
SELECT unique item_num, order_num, 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
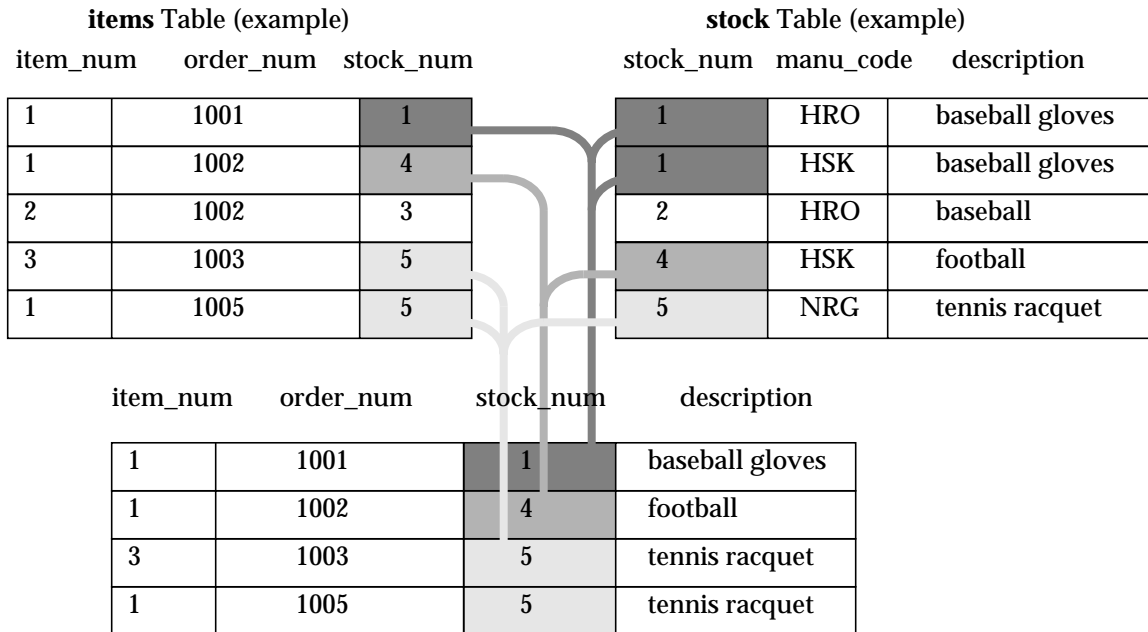
Figure 2-1   **Example of a join between two tables**

The following SELECT statement joins the **customer** and **state** tables.

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

The result of this query is composed of specified rows and columns from both the **customer** and **state** tables.

```
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
```

What This Chapter Contains

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 the next chapter.

Most of the examples in this chapter are taken from the nine tables in the **stores5** demonstration database, which is installed with the software for your Informix application development tool. In the interest of brevity, the examples may show only part of the data that is retrieved for each SELECT statement. See Chapter 1 in *The Informix Guide to SQL: Reference* for information on the structure and contents of the **stores5** database. For emphasis, keywords are shown in uppercase letters in the examples, although SQL is not case sensitive.

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 display the SELECT statements and their output as they appear when you use the interactive Query-Language option of **DB-Access** or **INFORMIX-SQL**. You also can use SELECT statements to query on data noninteractively through **INFORMIX-SQL** reports, you can embed them in a language such as **INFOR-MIX-ESQL/C** (where they are treated as executable code), or you can incorporate them in **INFORMIX-4GL** as part of its fourth-generation language.

Special Data Types

The examples use the **INFORMIX-OnLine** database server, which enables database applications to include the data types VARCHAR, TEXT, and BYTE. These data types are not available to applications that run on **INFORMIX-SE**.

In the **DB-Access** or **INFORMIX-SQL** Interactive Editor, when you issue a SELECT statement that includes one of these three data types, 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. For additional information on these and other data types, see Chapter 9 in this manual and Chapter 3 in *The Informix Guide to SQL: Reference.*

Single-Table SELECT Statements

There are many ways to query on a single table in a database. You can tailor a SELECT statement to

- 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 just the two required clauses, SELECT and FROM.

Using the Asterisk

The following statement 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.

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

The following statement uses the *wildcard* * as shorthand for the select list. The asterisk stands for the names of all the columns in the table. You can use the asterisk whenever you want all the columns, in their defined order.

```
SELECT * FROM manufact
```

The two examples are equivalent and display the same results, that is, a list of every column and row in the **manufact** table. The results are displayed as they would appear in the **DB-Access** or **INFORMIX-SQL** Interactive Editor.

```
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

You can change the order in which the columns are listed by changing their order in your select list.

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

This select list includes the same columns as the previous one, but because the columns are specified in a different order, the display is different also.

```
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 direct the system to sort the data in a specific order by adding an ORDER BY clause to your SELECT statement. The columns you want to use in the ORDER BY clause must be included in the select list either explicitly or implicitly.

An *explicit* select list includes all the column names.

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

An *implicit* select list uses the * symbol.

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

Either of the preceding statements produce the same display, that is, a list of every column and row in the **manufact** table, in order of **lead_time**:

```
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 number data types. DATE and DATETIME type data are sorted from earliest to latest, and INTERVAL data is ordered from shortest to longest span of time.

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

Descending Order

The keyword DESC following a column name causes the retrieved data to be sorted in *descending* order.

```
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 will sort the data based on the values in that column.

Sorting on Multiple Columns

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

In the following two examples of a nested sort, the order in which selected data is displayed is modified by changing the order of the two columns named in the ORDER BY clause.

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

Here, the **manu_code** column data is shown 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 of price.

```
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
```

In the following example, the order of the columns in the ORDER BY clause is reversed:

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

Here, the data is shown 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.

```
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 following four SELECT statements contain the same components in the ORDER BY clause, each produces a different result (not shown).

```
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.

Suppose you want to find all the customer numbers in the **orders** table.

```
SELECT customer_num FROM orders
```

This 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.

```
customer_num

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

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

You can cause duplicate rows to be suppressed by including the keyword DISTINCT or its synonym UNIQUE at the start of the select list.

```
SELECT DISTINCT customer_num FROM orders

SELECT UNIQUE customer_num FROM orders
```

Either of these statements limits the display to show each customer number in the **orders** table only once, producing a more readable list.

```
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. You decide to list all the purchase order numbers in the **orders** table.

```
SELECT po_num FROM orders
```

This SELECT statement retrieves data in the **po_num** column in the **orders** table and displays the following list:

```
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.**

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

```
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. The order in which the columns are selected is the order in which they are produced, from left to right.

```
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 earlier, you can use the ORDER BY clause to sort the data in ascending or descending order and perform nested sorts.

```
paid_date   ship_date  order_date customer_num   order_num po_num

            05/30/1991 05/22/1991         106       1004 8006
                       05/30/1991         112       1006 Q13557
            06/05/1991 05/31/1991         117       1007 278693
            06/29/1991 06/18/1991         117       1012 278701
            07/12/1991 06/29/1991         119       1016 PC6782
            07/13/1991 07/09/1991         120       1017 DM354331
06/03/1991  05/26/1991 05/21/1991         101       1002 9270
06/14/1991  05/23/1991 05/22/1991         104       1003 B77890
06/21/1991  06/09/1991 05/24/1991         116       1005 2865
07/10/1991  07/03/1991 06/25/1991         106       1014 8052
07/21/1991  07/06/1991 06/07/1991         110       1008 LZ230
07/22/1991  06/01/1991 05/20/1991         104       1001 B77836
07/31/1991  07/10/1991 06/22/1991         104       1013 B77930
08/06/1991  07/13/1991 07/10/1991         121       1018 S22942
08/06/1991  07/16/1991 07/11/1991         122       1019 Z55709
08/21/1991  06/21/1991 06/14/1991         111       1009 4745
08/22/1991  06/29/1991 06/17/1991         115       1010 429Q
08/22/1991  07/25/1991 07/23/1991         124       1021 C3288
08/22/1991  07/30/1991 07/24/1991         127       1023 KF2961
08/29/1991  07/03/1991 06/18/1991         104       1011 B77897
08/31/1991  07/16/1991 06/27/1991         110       1015 MA003
09/02/1991  07/30/1991 07/24/1991         126       1022 W9925
09/20/1991  07/16/1991 07/11/1991         123       1020 W2286
```

When you SELECT and ORDER BY several columns in a table, you might find it easier to use integers to refer to the position of the columns in the ORDER BY clause.

```
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
```

These two statements retrieve and display the same data:

```
customer_num    order_num po_num      order_date

       104         1001 B77836       05/20/1991
       101         1002 9270         05/21/1991
       104         1003 B77890       05/22/1991
       106         1004 8006         05/22/1991
       116         1005 2865         05/24/1991
       112         1006 Q13557       05/30/1991
       117         1007 278693       05/31/1991
       110         1008 LZ230        06/07/1991
       111         1009 4745         06/14/1991
       115         1010 429Q         06/17/1991
       104         1011 B77897       06/18/1991
       117         1012 278701       06/18/1991
       104         1013 B77930       06/22/1991
       106         1014 8052         06/25/1991
       110         1015 MA003        06/27/1991
       119         1016 PC6782       06/29/1991
       120         1017 DM354331     07/09/1991
       121         1018 S22942       07/10/1991
       122         1019 Z55709       07/11/1991
       123         1020 W2286        07/11/1991
       124         1021 C3288        07/23/1991
       126         1022 W9925        07/24/1991
       127         1023 KF2961       07/24/1991
```

You can include the DESC keyword in the ORDER BY clause when you have assigned integers to column names.

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

Here, data is sorted first in descending order by **order_date** and, within date, in ascending order by **customer_num**.

Selecting Substrings

You can select part of the value of a CHARACTER column by including 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.

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

This SELECT statement 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.

```
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

Perhaps you want to see only those orders placed by a particular customer or the calls entered by a particular customer service representative. You would add a WHERE clause to a SELECT statement to retrieve these specific rows from a table.

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 the *active set* for that statement. A *singleton* SELECT statement returns a single row. In **INFORMIX-4GL** or an embedded-language program, the retrieval of multiple rows requires the use of a *cursor.* See Chapters 6 and 7 in this manual.

Creating a Comparison Condition

The WHERE clause of a SELECT statement specifies the rows 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.

Figure 2-2 lists the *relational operators* 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

Figure 2-2   ***Relational operators that test for equality***

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 Chapter 7 of *The Informix Guide to SQL: Reference.* For DATE and DATETIME expressions, "greater than"

means *later in time*, and for INTERVAL expressions, it means *of longer duration.* 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

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

You also can use the preceding keywords and operators in a WHERE clause to create comparison condition queries that perform variable text searches using

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

The examples that follow illustrate these types of queries.

Including Rows

Use the relational operator = to include rows in a WHERE clause.

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

This SELECT statement returns the following set of rows:

```
customer_num call_code call_dtime        res_dtime

        106 D         1991-06-12 08:20 1991-06-12 08:25
        121 O         1991-07-10 14:05 1991-07-10 14:06
        127 I         1991-07-31 14:30
```

Excluding Rows

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

The following examples assume that you are selecting from an ANSI-compliant database, and thus they 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, although it is not incorrect to include it in either case. For a complete discussion of *owner naming,* see Chapter 7 in *The Informix Guide to SQL: Reference.*

```
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 of these statements exclude values by specifying that, in the **customer** table owned by the user **odin**, the value in the **state** column should not be equal to CA.

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 Rows

There are several ways to specify rows in a WHERE clause. Here are two:

```
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
```

Both of these statements specify a range for **catalog_num** from 10005 through 10008, inclusive. The first statement uses keywords, and the second uses relational operators to retrieve the following rows:

```
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
               Athletes

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
```

Note that 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 display TEXT and BYTE values by using the PROGRAM attribute when using forms in **INFORMIX**-**SQL** or **INFORMIX**-**4GL** or by writing a **4GL** or embedded-language program to do so.

Excluding a Range of Rows

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

Here, by using the keywords NOT BETWEEN, the condition excludes rows that have the character range 94000 through 94999 in the **zipcode** column.

```
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 earlier, these examples also assume the use of an ANSI-compliant database. Here, the owner qualifier is in quotation marks to preserve the case.

```
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
```

Both of these statements retrieve rows that include the subset of AZ or NJ in the **state** column of the **Aleta.customer** table.

```
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
```

Note that you cannot test a TEXT or BYTE column with the IN keyword.

In this example of a query on an ANSI-compliant database, there are no quotation marks around the table owner name. Whereas the two previous queries searched the table **Aleta.customer**, this SELECT statement searches the table **ALETA.customer** (which is a different table).

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

Here, by adding the keyword NOT IN, the subset is changed to exclude the subsets AZ and NJ in the **state** column. The results are in order of **state**.

```
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 represents either a value that is unknown or not applicable, or the absence of data. A NULL value is not the same as a zero or blank.

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

This SELECT statement returns all rows that have a NULL **paid_date**.

```
    order_num   customer_num   po_num     ship_date

        1004            106   8006       05/30/1991
        1006            112   Q13557
        1007            117   278693     06/05/1991
        1012            117   278701     06/29/1991
        1016            119   PC6782     07/12/1991
        1017            120   DM354331   07/13/1991
```

Forming Compound Conditions

You can connect two or more comparison conditions or *Boolean* expressions by using 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.

Here, the operator AND combines two comparison expressions in the WHERE clause.

```
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
```

This SELECT statement returns all rows that have a NULL **paid_date** *and* that do not also have a NULL **ship_date.**

```
order_num customer_num po_num      ship_date

    1004          106 8006         05/30/1991
    1007          117 278693       06/05/1991
    1012          117 278701       06/29/1991
    1016          119 PC6782       07/12/1991
    1017          120 DM354331     07/13/1991
```

Using Variable Text Searches

You can use the keywords LIKE and MATCHES for *variable text* queries based on substring searches of CHARACTER 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 in Figure 2-3 with LIKE or MATCHES.

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 single character or range of values
\	escapes special significance of next character

Figure 2-3   Wildcards used with LIKE and MATCHES

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 relational operator =. Unlike earlier examples, these examples illustrate how to query on an *external* table in an ANSI-compliant database.

An external table is a table that is not in the current database. You can access only external tables that are part of an ANSI-compliant database.

Whereas the database used previously in this chapter was the demonstration database called **stores5**, 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 defining external tables, see Chapter 7 in *The Informix Guide to SQL: Reference.*

```
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
```

Any of these SELECT statements retrieves all the rows that have the single word helmet in the **description** column.

```
  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 following examples illustrate the use of a single-character wildcard in a WHERE clause. Further, they demonstrate a query on an external table. Here, the **stock** table is in the external database **sloth**, which, besides being outside the current **stores5** database, is on a separate database server called **meerkat**.

For details on external tables, external databases, and networks, see Chapter 12 in this manual and Chapter 7 in *The Informix Guide to SQL: Reference.*

```
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
```

Both SELECT statements retrieve only those rows for which the middle letter of the **manu_code** is R.

```
stock_num manu_code description      unit_price unit unit_descr

    205 HRO       3 golf balls     $312.00 each 24/case
    306 PRC       Tandem adapter   $160.00 each each
    307 PRC       baby jogger      $250.00 each each
      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
    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

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

This statement selects only those rows where the **manu_code** begins with A through H and returns the following rows:

```
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
```

The class test "[A-H]" specifies any single letter from A through H inclusive. There is no equivalent wildcard symbol for the LIKE keyword.

WHERE Clause with Variable-Length Wildcard

These examples use a wildcard at the end of a string to retrieve all the rows where the **description** begins with the characters bicycle.

```
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 SELECT statement returns these rows:

```
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 char-
acters `bicycle` alone, if there is a row with that description.

The next SELECT statement narrows the search further by adding another
comparison condition that excludes a **manu_code** of PRC.

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

The SELECT statement retrieves only these rows:

```
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
```

Note that when you select from a large table and use an initial wildcard in the
comparison string, the query often takes longer to execute. This is because
indexes cannot be used, so every row must be searched.

Comparing for Special Characters

The keyword ESCAPE used with LIKE or MATCHES lets you protect a special
character from misinterpretation as a wildcard symbol.

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

The ESCAPE keyword designates an *escape character* (it is ! in this example), which protects one following character so 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 with the LIKE wildcard %. The query retrieves the following row:

```
customer_num    116
call_dtime      1990-12-21 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       1990-12-27 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.

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

The subscript [1,4] causes the query to retrieve all rows in which the first four letters of the **cat_advert** column are High.

```
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.
```

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. You do this by listing 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 connected by operators. It also can include host variables (program data) when the SELECT statement is embedded in a program.

Arithmetic Expressions

An arithmetic expression contains one or more *arithmetic operators* listed in Figure 2-4 and translates to a number.

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division

Figure 2-4   Arithmetic operators used in arithmetic expressions

Operations of this nature enable you to see the results of proposed computations without actually altering the data in the database. Add an INTO TEMP clause to save the altered data in a temporary table for further reference, computations, or impromptu reports.

You cannot use TEXT or BYTE columns in arithmetic expressions.

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

This SELECT statement calculates a 7% sales tax on the **unit_price** column when the **unit_price** is $400 or higher (but does not update it in the database). If you are using the **DB**-**Access** or **INFORMIX**-**SQL** Interactive Editor, the result is displayed in a column labeled *expression*.

```
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
```

This statement calculates a surcharge of $6.50 on orders when the quantity ordered is less than 5.

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

If you are using **DB-Access** or **INFORMIX-SQL**, the result is displayed in a column labeled *expression.*

```
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
```

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

This SELECT statement calculates and displays in an *expression* column (if you are using **DB**-**Access** or **INFORMIX**-**SQL**) the interval between when the customer call was received (**call_dtime**) and when the call was resolved (**res_dtime**), in days, hours, and minutes.

```
customer_num user_id          call_code call_dtime          (expression)

         116 mannyn           I         1990-12-21 11:24       5 20:55
         116 mannyn           I         1990-11-28 13:34       0 03:13
         106 maryj            D         1991-06-12 08:20       0 00:05
         121 maryj            O         1991-07-10 14:05       0 00:01
         127 maryj            I         1991-07-31 14:30
         110 richc            L         1991-07-07 10:24       0 00:06
         119 richc            B         1991-07-01 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 the three preceding examples, you can define a display label after the computation in the SELECT clause. The results displayed will be the same as shown earlier, but the column displaying derived values will now have descriptive headers.

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

Here, the label **taxed** is assigned to the expression in the select list that displays the results of the operation unit_price * 1.07.

```
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 this SELECT statement, the label **surcharge** is defined for the column that displays the results of the operation `total_price + 6.50`.

```
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.

```
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
     .
     .
     .
```

This SELECT statement assigns the label **span** to the column that displays the results of subtracting the DATETIME column **call_dtime** from the DATETIME column **res_dtime**.

```
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.

```
customer_num user_id           call_code call_dtime           span

        116 mannyn            I          1990-12-21 11:24    5 20:55
        116 mannyn            I          1990-11-28 13:34    0 03:13
        106 maryj             D          1991-06-12 08:20    0 00:05
        121 maryj             O          1991-07-10 14:05    0 00:01
        127 maryj             I          1991-07-31 14:30
        110 richc             L          1991-07-07 10:24    0 00:06
        119 richc             B          1991-07-01 15:00    0 17:21
```

Sorting on Derived Columns

When you want to ORDER BY an expression, you can use either the display label assigned to the expression or an integer.

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

This query retrieves the same data from the **cust_calls** table as the previous query, but here the ORDER BY clause causes the data to be displayed in ascending order of the derived values in the **span** column.

```
customer_num user_id           call_code call_dtime              span

        127 maryj             I          1991-07-31 14:30
        121 maryj             O          1991-07-10 14:05    0 00:01
        106 maryj             D          1991-06-12 08:20    0 00:05
        110 richc             L          1991-07-07 10:24    0 00:06
        116 mannyn            I          1990-11-28 13:34    0 03:13
        119 richc             B          1991-07-01 15:00    0 17:21
        116 mannyn            I          1990-12-21 11:24    5 20:55
```

The next version uses an integer to represent the result of the operation res_dtime - call_dtime and retrieves the same rows.

```
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 also can include one or more functions.

There are five *aggregate* functions and eight *time* functions, as well as the LENGTH, USER, TODAY, HEX, ROUND, and TRUNC functions. These functions are described in detail in Chapter 7 of *The Informix Guide to SQL: Reference.*

Aggregate Functions

The *aggregate* functions are COUNT, AVG, MAX, MIN, and SUM. They take on values that depend on all the rows selected and return information about rows, not the rows themselves. You cannot use these functions with TEXT or BYTE columns.

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

This SELECT statement counts and displays the total number of rows in the **stock** table.

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

This is the result:

```
     (count(*))

            73
```

This statement 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.

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

This is the result:

```
     (count(*))

            16
```

By including the keyword DISTINCT (or its synonym UNIQUE) and a column name in the SELECT statement, you can tally the number of different manu-facturer codes in the **stock** table.

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

This is the result:

```
     (count)

          9
```

This SELECT statement computes the average **unit_price** of all rows in the **stock** table.

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

This is the result:

```
      (avg)

     $196.00
```

This SELECT statement computes the average **unit_price** of just those rows in the **stock** table that have a **manu_code** of SHM.

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

This is the result:

```
      (avg)

     $200.24
```

You can combine aggregate functions in a SELECT statement.

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

This SELECT statement finds and displays both the highest and lowest **ship_charge** in the **orders** table.

```
  (max)      (min)

 $25.20     $5.00
```

You can apply functions to expressions, and you can supply display labels for their results.

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

This query 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.

```
 maximum          minimum          average

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

```
SELECT SUM (ship_weight)
    FROM orders
    WHERE ship_date = "07/13/1991"
```

This SELECT statement calculates and displays the total **ship_weight** of orders shipped on July 13, 1991.

```
 (sum)

      130.5
```

Time Functions

You can use the *time* functions DAY, MDY, MONTH, WEEKDAY, YEAR, and DATE 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 also can use the CURRENT function to return a value with the current date and time, or the EXTEND function to adjust the precision of a DATE or DATETIME value.

Using DAY and CURRENT

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

This SELECT statement returns the day of the month for the **call_dtime** and **res_dtime** columns in two *expression* columns.

```
customer_num (expression) (expression)

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

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

This SELECT statement 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.

```
customer_num (expression) (expression)
         106           12          12
         110            7           7
         119            1           2
         121           10          10
```

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

This query shows another use of the CURRENT function, selecting rows where the day is earlier than the current one.

```
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 MONTH

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

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

```
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
```

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

This SELECT statement 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.

```
customer_num        called      resolved

       106             6            6
       110             7            7
       119             7            7
       121             7            7
```

Using WEEKDAY

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

In this SELECT statement, the WEEKDAY function is used to indicate which day of the week calls were received and resolved (0 represents Sunday, 1 is Monday, and so on), and the expression columns are labeled.

```
customer_num called resolved

        127      2
        119      0         1
        106      2         2
        121      2         2
        116      2         2
        116      4         3
        110      6         6
```

```
SELECT COUNT(*)
    FROM cust_calls
    WHERE WEEKDAY (call_dtime) IN (0,7)
```

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

```
        (count(*))

            1
```

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

This SELECT statement retrieves rows where the **call_dtime** is earlier than the beginning of the current year.

```
  customer_num call_code     call_year      res_year

         116 I                    1990          1990
         116 I                    1990          1990
```

Formatting DATETIME Values

In the following query, the EXTEND function restricts the two DATETIME values by displaying only the specified subfields.

```
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
```

The result returns the month-to-minute range for the columns labeled **call_time** and **res_time** and gives an indication of the workload.

```
  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
```

Using the DATE Function

This query retrieves DATETIME values only when **call_dtime** is later than the specified DATE.

```
SELECT customer_num, call_dtime, res_dtime
    FROM cust_calls
    WHERE call_dtime > DATE ("12/31/90")
```

It returns these rows:

```
customer_num call_dtime        res_dtime

        106 1991-06-12 08:20 1991-06-12 08:25
        110 1991-07-07 10:24 1991-07-07 10:30
        119 1991-07-01 15:00 1991-07-02 08:21
        121 1991-07-10 14:05 1991-07-10 14:06
        127 1991-07-31 14:30
```

```
SELECT customer_num,
    DATE (call_dtime) called,
    DATE (res_dtime) resolved
    FROM cust_calls
    WHERE call_dtime >= DATE ("1/1/91")
```

Here, the SELECT statement 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.

```
customer_num      called      resolved

        106 06/12/1991   06/12/1991
        110 07/07/1991   07/07/1991
        119 07/01/1991   07/02/1991
        121 07/10/1991   07/10/1991
        127 07/31/1991
```

Other Functions and Keywords

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

You can use these functions and keywords to select an expression that consists entirely of constant values or one 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.

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

In this SELECT statement, 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.

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

While perhaps not too useful when you work with the **DB**-**Access** or **INFORMIX**-**SQL** Interactive Editor, the LENGTH function 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.

The USER function can be handy when you want to define a restricted view of a table that contains only your rows. For information on creating views, see Chapter 11 in this manual and the GRANT and CREATE VIEW statements in Chapter 7 of *The Informix Guide to SQL: Reference.*

This SELECT statement specifies the **cust_calls** table.

```
SELECT USER from cust_calls
```

This SELECT statement returns the user name (login account name) of the user who executes the query. It is repeated once for each row in the table.

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

If the user name of the current user is **richc**, this SELECT statement retrieves only those rows in the **cust_calls** table that are owned by that user.

```
customer_num  110
call_dtime    1991-07-07 10:24
user_id       richc
call_code     L
call_descr    Order placed one month ago (6/7) not received.
res_dtime     1991-07-07 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    1991-07-01 15:00
user_id       richc
call_code     B
call_descr    Bill does not reflect credit from previous order
res_dtime     1991-07-02 08:21
res_descr     Spoke with Jane Akant in Finance. She found the
    error and is sending new bill to customer
```

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

This SELECT statement, if issued when today's system date is July 10, 1991, returns this one row:

```
order_num      1018
order_date     07/10/1991
customer_num   121
ship_instruct  SW corner of Biltmore Mall
backlog        n
po_num         S22942
ship_date      07/13/1991
ship_weight    70.50
ship_charge    $20.00
paid_date      08/06/1991
```

You can include the keyword SITENAME in a SELECT statement on **INFOR-MIX**-**OnLine** to find the name of the database server. You can query on the SITENAME for any table that has rows, including system catalog tables.

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

In this SELECT statement, you assign the label **server** to the SITENAME 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.

```
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.

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

In this SELECT statement, the HEX function returns the hexadecimal format of three specified columns in the **customer** table:

```
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
```

Multiple-Table SELECT Statements

You can select data from two or more tables by naming 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 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 will be better when the columns in the join condition are indexed.

Data types are described in Chapter 3 of *The Informix Guide to SQL: Reference*, and indexing is discussed in Chapters 4 and 10 of this manual.

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 is usually a very large, unwieldy result and the data will not be accurate.

This SELECT statement selects from two tables and produces a Cartesian product.

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

Although there are only 52 rows in the **state** table and 28 rows in the **customer** table, the effect of this SELECT statement is to multiply the rows of one table by the rows of the other and retrieve an impractical 1456 rows.

```
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
.
.
.
```

Note that some of the data displayed in the concatenated rows is inaccurate. For example, while 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*. More complex forms, such as *self-joins* and *outer joins,* are covered in Chapter 3.

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.

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

This SELECT statement joins the **manufact** and **stock** tables on the **manu_code** column, retrieving only those rows for which the values for the two columns are equal.

```
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
 .
 .
 .
```

Note that in this equi-join, the output includes the **manu_code** column from
both the **manufact** and **stock** tables because the select list requested every
column.

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

When columns in the joined tables have the same name, the columns must be preceded by the name of a specific table and a period, as shown in the following example:

```
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
```

This SELECT statement joins on 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. It returns these rows:

```
order_num       1004
order_date      05/22/1991
ship_date       05/30/1991
customer_num    106
call_dtime      1991-06-12 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       1991-06-12 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      06/07/1991
ship_date       07/06/1991
customer_num    110
call_dtime      1991-07-07 10:24
user_id         richc
call_code       L
call_descr      Order placed one month ago (6/7) not received.
res_dtime       1991-07-07 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      07/24/1991
ship_date       07/30/1991
customer_num    127
call_dtime      1991-07-31 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 in the following example:

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

Like the previous example, this SELECT statement joins the **manufact** and **stock** tables on the **manu_code** column, but because the select list is more closely defined, the **manu_code** is listed only once for each row retrieved:

```
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.

```
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
```

Both of these SELECT statements create the same natural join and retrieve the following row:

```
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
```

Note that this display 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.

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

This SELECT statement creates an equi-join on the **catalog**, **stock**, and **manufact** tables and retrieves the following row:

```
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
```

Note that the **manu_code** is repeated three times, once for each table, and **stock_num** is repeated twice.

Because of the considerable duplication in the previous example of a multiple-table query, it is wise to more closely define the SELECT statement by including specific columns in the select list.

```
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
```

This SELECT statement uses a wildcard to select all columns from the table having the most columns and then specifies columns from the other two tables. This SELECT statement produces the following natural join that displays the same information as the previous example, but without duplication:

```
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 produce output for other uses.

Using Aliases

You can make multiple-table queries shorter and more readable by assigning *aliases* to the tables in a SELECT statement. 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.

```
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 this example, 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 the preceding SELECT statement to the following one, which does not use aliases.

```
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
```

These two SELECT statements are equivalent and retrieve the following data:

```
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
```

Note that you cannot ORDER BY the TEXT column **cat_descr** or the BYTE column **cat_picture**.

You also can use aliases to shorten your queries on external tables residing in external databases.

```
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
```

This SELECT statement joins columns from two tables that reside in different databases and systems, neither of which is the current database or system. 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 this information.

```
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 external tables and external databases, see Chapter 12 in this manual and Chapter 7 in *The Informix Guide to SQL: Reference.*

You also can use *synonyms* as shorthand references to the long names of external and current tables and views. For details on how to create and use synonyms, see the CREATE SYNONYM statement in Chapter 7 of *The Informix Guide to SQL: Reference.*

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 can be queried or manipulated without modifying the database. Temporary tables are dropped when you end your SQL session or when your program or report terminates.

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

This SELECT statement creates a temporary table called **stockman** and stores the results of the query in it.

```
stock_num manu_name      description      unit_price (expression)

        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
```

You can query on this table and join it with other tables, thus avoiding a multiple sort and moving more quickly through the database. Temporary tables are discussed at greater length in Chapter 4 of this manual.

Summary

This chapter introduced sample syntax and results for basic kinds of SELECT statements that are used to query on a relational database. Earlier sections of the chapter showed how to perform the following activities:

- 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.
- 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.
- Perform variable text searches using exact text comparisons, variable-length wildcards, and restricted and unrestricted wildcards.
- 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 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 COUNT, AVG, MAX, MIN, and SUM 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.

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" told how to

- Create a Cartesian product.
- Constrain a Cartesian product by including a WHERE clause with a valid join condition in your query.
- 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.

Advanced SELECT Statements

3

Chapter Overview

The preceding chapter demonstrated some of the 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 returned by the GROUP BY clause.

This chapter 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*, where 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 state-
ment clauses in your queries. The clauses must appear in this order:

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

An additional SELECT statement clause, INTO, which you can use to specify
program and host variables in **INFORMIX-4GL** and the embedded-language
products, is described in Chapter 6 of this manual, as well as the product
manuals.

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 they are
formed. 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. It is most often combined with
aggregate functions that produce summary values for each of those sets.
Some of the examples in Chapter 2 showed 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 included this example:

```
SELECT DISTINCT customer_num FROM orders
```

You also could write it this way:

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

Either statement returns these rows:

```
customer_num

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

The GROUP BY clause collected the rows into sets so that the rows in each set all had 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 it is used with aggregate functions.

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

This SELECT statement retrieves the number of items and the total price of all items for each order. 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 the groups are formed, the aggregate functions COUNT and SUM are applied within each group.

The query returns one row for each group. It uses labels to give names to the results of the COUNT and SUM expressions as follows:

```
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
```

The SELECT statement collected the rows of the **items** table into groups that had identical order numbers and computed the COUNT of rows in each group and the sum of the prices.

Note that you cannot include a column having a TEXT or BYTE data type in a GROUP BY clause. To *group*, you must be able to *sort*, and there is no natural sort order 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 to sort on an aggregate in the select list.

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

This query is the same as the previous one but includes an ORDER BY clause to sort the retrieved rows in ascending order of **price**:

```
    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 the preceding chapter, 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.

This SELECT statement returns the same rows as the previous query:

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
    FROM items
    GROUP BY 1
    ORDER BY 3
```

When you build your SELECT statement, remember that all non-aggregate 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 for this is that a SELECT with GROUP BY 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.

You can use the GROUP BY clause in a SELECT statement that joins tables.

```
SELECT o.order_num, SUM (i.total_price)
    FROM orders o, items i
    WHERE o.order_date > "01/01/90"
        AND o.customer_num = 110
        AND o.order_num = i.order_num
    GROUP BY o.order_num
```

This query joins the **orders** and **items** tables, assigns table aliases to them, and returns the following two rows:

```
order_num              (sum)

    1008          $940.00
    1015          $450.00
```

Using the HAVING Clause

The HAVING clause usually complements a GROUP BY clause by applying one or more qualifying conditions to groups after they are formed, similar to the way the WHERE clause qualifies individual rows. One advantage to 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.

```
SELECT order_num, COUNT(*) number, AVG (total_price) average
    FROM items
    GROUP BY order_num
    HAVING COUNT(*) > 2
```

This SELECT statement 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 composed of two or more rows.

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.

```
SELECT AVG (total_price) average
    FROM items
    HAVING count(*) > 2
```

This statement, a modified version of the previous example, returns just one row, the average of all **total_price** values in the table:

```
average

$270.97
```

If this query, like the previous one, had included the non-aggregate column **order_num** in the select list, you would have had to include a GROUP BY clause with that column in the group list. In addition, if the condition in the HAVING clause had not been satisfied, the output would have shown the column heading and a message would have indicated that no rows were found.

The following example contains all seven SELECT statement clauses that you can use in the Informix version of interactive SQL (the INTO clause naming program or host variables is available only in an **INFORMIX-4GL** or embedded-language program):

```
SELECT o.order_num, SUM (i.total_price) price,
     paid_date - order_date span
   FROM orders o, items i
   WHERE o.order_date > "01/01/90"
       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
```

This SELECT statement joins the **orders** and **items** tables; employs display labels, table aliases, and integers used as column indicators; groups and orders the data; and puts the following results in a temporary table:

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

The previous chapter showed how to include a WHERE clause in a SELECT statement to join two or more tables on one or more columns. It illustrated natural joins and equi-joins.

This chapter discusses the uses of two more complex kinds of joins: self-joins and outer joins. As described for simple joins in Chapter 2, you can define aliases for tables and assign display labels to expressions to shorten your multiple-table queries. You also can sort data with an ORDER BY clause and SELECT query results 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*. This 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, assigning 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 different tables. (Aliases in SELECT statements are shown in Chapter 2 of this manual and discussed in Chapter 7 of *The Informix Guide to SQL: Reference.*)

Just as in joins between tables, you can use arithmetic expressions in self-joins. You can test for NULL values, and you can ORDER BY a specified column in ascending or descending order.

```
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
```

This SELECT statement finds pairs of orders where the **ship_weight** differs by a factor of five or more and the **ship_date** is not NULL, and orders the data by **ship_date**:

```
  order_num ship_weight ship_date    order_num ship_weight ship_date

      1004       95.80 05/30/1991        1011       10.40 07/03/1991
      1004       95.80 05/30/1991        1020       14.00 07/16/1991
      1004       95.80 05/30/1991        1022       15.00 07/30/1991
      1007      125.90 06/05/1991        1015       20.60 07/16/1991
      1007      125.90 06/05/1991        1020       14.00 07/16/1991
      1007      125.90 06/05/1991        1022       15.00 07/30/1991
      1007      125.90 06/05/1991        1011       10.40 07/03/1991
      1007      125.90 06/05/1991        1001       20.40 06/01/1991
      1007      125.90 06/05/1991        1009       20.40 06/21/1991
      1005       80.80 06/09/1991        1011       10.40 07/03/1991
      1005       80.80 06/09/1991        1020       14.00 07/16/1991
      1005       80.80 06/09/1991        1022       15.00 07/30/1991
      1012       70.80 06/29/1991        1011       10.40 07/03/1991
      1012       70.80 06/29/1991        1020       14.00 07/16/1991
      1013       60.80 07/10/1991        1011       10.40 07/03/1991
      1017       60.00 07/13/1991        1011       10.40 07/03/1991
      1018       70.50 07/13/1991        1011       10.40 07/03/1991
       .
       .
       .
```

Suppose you want to select the results of a self-join into a temporary table. You would, of course, append an INTO TEMP clause to the SELECT statement. However, because you are, in effect, creating a new table, you also must rename at least one set of column names by assigning them display labels. Otherwise, you will see an error message that indicates duplicate column names, and the temporary table is not created.

```
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
    INTO TEMP shipping
```

This SELECT statement, which is similar to the previous one, labels all columns selected from the **orders** table and puts them in a temporary table called **shipping**. If you SELECT * from that table, you see these rows:

orders1	purch1	ship1	orders2	purch2	ship2
1004	8006	05/30/1991	1011	B77897	07/03/1991
1004	8006	05/30/1991	1020	W2286	07/16/1991
1004	8006	05/30/1991	1022	W9925	07/30/1991
1005	2865	06/09/1991	1011	B77897	07/03/1991
1005	2865	06/09/1991	1020	W2286	07/16/1991
1005	2865	06/09/1991	1022	W9925	07/30/1991
1007	278693	06/05/1991	1001	B77836	06/01/1991
1007	278693	06/05/1991	1009	4745	06/21/1991
1007	278693	06/05/1991	1011	B77897	07/03/1991
1007	278693	06/05/1991	1015	MA003	07/16/1991
1007	278693	06/05/1991	1020	W2286	07/16/1991
1007	278693	06/05/1991	1022	W9925	07/30/1991
1012	278701	06/29/1991	1011	B77897	07/03/1991
1012	278701	06/29/1991	1020	W2286	07/16/1991
1013	B77930	07/10/1991	1011	B77897	07/03/1991
1017	DM354331	07/13/1991	1011	B77897	07/03/1991
1018	S22942	07/13/1991	1011	B77897	07/03/1991
1018	S22942	07/13/1991	1020	W2286	07/16/1991
1019	Z55709	07/16/1991	1011	B77897	07/03/1991
1019	Z55709	07/16/1991	1020	W2286	07/16/1991
1019	Z55709	07/16/1991	1022	W9925	07/30/1991
1023	KF2961	07/30/1991	1011	B77897	07/03/1991

You can join a table to itself more than once. The maximum number of self-joins depends on the resources available to you.

```
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
```

This self-join creates a list of those items in the **stock** table that are supplied by three manufacturers. By including the last two conditions in the WHERE clause, it eliminates duplicate manufacturer codes in rows retrieved.

```
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
```

Say you want to select rows from a payroll table to determine which employees earn more than their manager. You can construct the following self-join:

```
SELECT emp.employee_num, emp.gross_pay, emp.level,
      emp.dept_num, mgr.employee_num, mgr.gross_pay,
      mgr.level
    FROM payroll emp, payroll mgr
    WHERE emp.gross_pay > mgr.gross_pay
        AND emp.level < mgr.level
    ORDER BY 4
```

The following example of a self-join uses a *correlated subquery* to retrieve and list the 10 highest-priced items ordered:

```
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
```

It returns the following 10 rows:

```
    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 this chapter.

Using Rowid Values

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."

```
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
```

This SELECT statement selects data twice from the **cust_calls** table, assigning it the table aliases **x** and **y**. It searches for duplicate values in the **customer_num** column, and for their rowids, finding this pair:

```
        rowid customer_num

        515          116
        769          116
```

You can write the last condition in the previous SELECT statement either as

```
AND x.rowid != y.rowid
```

or

```
AND NOT x.rowid = y.rowid
```

Another way to locate duplicate values is with a correlated subquery, as follows:

```
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)
```

This SELECT statement locates the same two duplicate **customer_num** values as the previous query and returns these rows:

```
customer_num call_dtime

        116 1990-11-28 13:34
        116 1990-12-21 11:24
```

You can use the rowid, shown earlier in a self-join, to locate the internal record number associated with a row in a database table. Rowid is, in effect, a hidden column in every table. The sequential values of rowid have no special significance and may vary depending on the location of the physical data in the chunk. See Chapter 4 in this manual for a discussion of performance issues and the rowid value.

```
SELECT rowid, * FROM manufact
```

This SELECT statement uses rowid and the wildcard * in the SELECT clause to retrieve every row in the **manufact** table and their corresponding rowids.

```
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 rowid when you select a specific column.

```
SELECT rowid, manu_code FROM manufact
```

This SELECT statement produces these results:

```
rowid manu_code

  258 ANZ
  261 HRO
  260 HSK
  263 KAR
  264 NKL
  259 NRG
  265 PRC
  262 SHM
  257 SMT
```

You also 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.

```
SELECT * FROM manufact WHERE rowid = 263
```

This SELECT statement returns just one row:

```
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 and, on **INFORMIX**-**OnLine**, with the SITENAME keyword. USER and SITENAME were discussed in Chapter 2 of this manual.

```
SELECT USER username, rowid FROM cust_calls
```

This query assigns the label **username** to the USER expression column and returns this information about the **cust_calls** table:

```
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.

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

This query returns the rowid for only those rows entered by the user who entered the statement. For example, if the user **richc** entered this SELECT statement, this would be the output:

```
   rowid

     258
     259
```

Using the SITENAME Function

Add the DBSERVERNAME keyword to a query to find out where the current database resides.

```
SELECT DBSERVERNAME server, tabid, rowid, USER username
    FROM systables
    WHERE tabid >= 105 OR rowid <= 260
    ORDER BY rowid
```

This SELECT statement finds the 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. It assigns display labels to the DBSERVERNAME and USER expressions and returns these 10 rows from the **systables** system catalog table:

```
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
```

Note that you should 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 will be different.

Outer Joins

Chapter 2 showed 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 *unsymmetrically*. It makes one of the tables *dominant* (also called "preserved") over the other *subservient* tables.

There are four basic types of outer joins:

- A simple outer join on two tables
- A simple outer join to a third table
- An outer join for a simple join to a third table
- An outer join for an outer join to a third table

These four types of outer joins are illustrated in this section. See the discussion of outer joins in Chapter 7 of *The Informix Guide to SQL: Reference* for full information on their syntax, use, and logic.

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. *Rows from the dominant table that would otherwise be discarded are preserved, even though no matching row was found in the subservient table.* The dominant-table rows that do not have a matching subservient-table row receive a row of nulls before the selected columns are projected.

An outer join applies conditions to the subservient table while sequentially applying 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. You transform a simple join into an outer join by inserting 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 make heavy use of outer joins, you should determine whether one or more simple joins will do. 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 the preceding chapter.

Simple Join

This is an example of the type of simple join on the **customer** and **cust_calls** tables shown in Chapter 2 of this manual:

```
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
```

This query returns only those rows where the customer has made a call to customer service:

```
customer_num  106
lname         Watson
company       Watson & Son
phone         415-389-8789
call_dtime    1991-06-12 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    1991-07-07 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    1991-07-01 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    1991-07-10 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    1991-07-31 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    1990-11-28 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    1990-12-21 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

The following example uses the same select list, tables, and comparison condition as the preceding example, but this time creates a simple outer join:

```
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 in front of 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 corresponding rows from the subservient **cust_calls** table.

```
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    1991-06-12 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

The following example 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*:

```
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
```

This SELECT statement 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 this form:

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 for an Outer Join to a Third Table

This SELECT statement creates an outer join that is the result of an outer join to a third table. This third type is known as a *nested outer join*:

```
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
```

This query 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 an outer join, which combines this information with data from the dominant **customer** table. This query preserves order numbers that the previous example eliminated, returning rows for orders that do not contain items with either manufacturer code. An optional ORDER BY clause reorganizes the data.

```
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
```

There are two ways to state the join conditions 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.

Outer Join of Two Tables to a Third Table

This SELECT statement shows an outer join that is the result of an outer join of each of two tables to a third table. In this fourth type of outer join, join relationships are possible *only* between the dominant table and the subservient tables:

```
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
   INTO TEMP service
```

This query 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.

```
customer_num lname          order_num order_date call_dtime

        114 Albertson
        118 Baxter
        113 Beatty
        103 Currie
        115 Grant             1010 06/17/1991
        123 Hanlon            1020 07/11/1991
        125 Henry
        104 Higgins           1003 05/22/1991
        104 Higgins           1001 05/20/1991
        104 Higgins           1013 06/22/1991
        104 Higgins           1011 06/18/1991
        110 Jaeger            1015 06/27/1991 1991-07-07 10:24
        110 Jaeger            1008 06/07/1991 1991-07-07 10:24
        120 Jewell            1017 07/09/1991
        111 Keyes             1009 06/14/1991
        112 Lawson            1006 05/30/1991
        109 Miller
        128 Moore
        126 Neelie            1022 07/24/1991
        122 O'Brian           1019 07/11/1991
        116 Parmelee          1005 05/24/1991 1990-12-21 11:24
        116 Parmelee          1005 05/24/1991 1990-11-28 13:34
        101 Pauli             1002 05/21/1991
        124 Putnum            1021 07/23/1991
        108 Quinn
        107 Ream
        102 Sadler
        127 Satifer           1023 07/24/1991 1991-07-31 14:30
        119 Shorter           1016 06/29/1991 1991-07-01 15:00
        117 Sipes             1007 05/31/1991
        117 Sipes             1012 06/18/1991
        105 Vector
        121 Wallack           1018 07/10/1991 1991-07-10 14:05
        106 Watson            1004 05/22/1991 1991-06-12 08:20
        106 Watson            1014 06/25/1991 1991-06-12 08:20
```

Note that if the preceding SELECT statement had tried to create a join condition between the two subservient tables **o** and **x**, as in the following example, an error message would have indicated the creation of a two-sided outer join:

```
WHERE o.customer_num = x.customer_num
```

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, and must be enclosed in parentheses that tell the database server to perform 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 produced by the outer SELECT. An uncorrelated subquery is executed only once.

Often, you can construct a SELECT statement with a subquery to replace two separate SELECT statements.

Subqueries in SELECT statements allow you to

- Compare an expression to the result of another SELECT statement.
- Determine whether an expression is included in the results of another SELECT statement.
- Determine whether any rows are selected by another SELECT statement.

An optional WHERE clause in a subquery often is 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 it returns *no* value, the query returns no rows at all. Such a subquery is equivalent to a NULL value.
- If it returns *one* value, the subquery returns either one aggregate expression or else selects exactly one row and one column. Such a subquery is equivalent to a single number or character value.
- If it returns a list or *set* of values, the subquery returns 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 of the relational operators 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 at all; that is, it asks if the result of the subquery is not null.

See Chapter 7 in *The Informix Guide to SQL: Reference* for the complete syntax used in creating a condition with a subquery. See also Chapter 4 of this manual for information on performance implications for correlated and uncorrelated subqueries.

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 at all, the condition is true of all of the zero values.)

```
SELECT order_num, stock_num, manu_code, total_price
    FROM items
    WHERE total_price < ALL
        (SELECT total_price FROM items
            WHERE order_num = 1023)
```

This SELECT statement 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:

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) preceding a subquery to deter-
mine whether a comparison is true for at least one of the values returned. If
the subquery returns no values, the search condition is *false.* (Since there were
no values, the condition could not be true of one of them.)

```
SELECT DISTINCT order_num
    FROM items
    WHERE total_price > ANY
        (SELECT total_price
            FROM items
            WHERE order_num = 1005)
```

This query 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, returning these rows:

```
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 sub-query will 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, since aggregate functions always return single values:

```
SELECT order_num FROM items
    WHERE stock_num = 9
        AND quantity =
            (SELECT MAX (quantity)
                FROM items
                WHERE stock_num = 9)
```

This SELECT statement uses the aggregate function MAX in a subquery to find the **order_num** for orders that include the maximum number of volleyball nets. It returns this row:

```
order_num

    1012
```

The following example uses the aggregate function MIN in the subquery to select items for which the total price is higher than 10 times the minimum price:

```
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)
```

The query retrieves these rows:

```
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

The following example of a correlated subquery returns a list of the 10 earliest 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.

```
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 it produces depends on **main.ship_date**, a value produced by the outer SELECT. Thus, the subquery must be executed anew for every row that the outer query considers.

The subquery 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 13 rows that have the 10 latest shipping dates.

```
po_num      ship_date

4745        06/21/1991
278701      06/29/1991
429Q        06/29/1991
8052        07/03/1991
B77897      07/03/1991
LZ230       07/06/1991
B77930      07/10/1991
PC6782      07/12/1991
DM354331    07/13/1991
S22942      07/13/1991
MA003       07/16/1991
W2286       07/16/1991
Z55709      07/16/1991
C3288       07/25/1991
KF2961      07/30/1991
W9925       07/30/1991
```

If you use a correlated subquery like the preceding one on a very large table, you should index the **ship_date** column to improve performance. Otherwise, this SELECT statement might be considered somewhat inefficient because it executes the subquery once for every row of the table. Indexing and performance issues are discussed in Chapter 10 of this manual.

Using EXISTS

The keyword EXISTS is known as an *existential qualifier* because the subquery is true only if the outer SELECT finds at least one row.

```
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. You also can substitute =ANY for IN.

```
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
```

Both of the preceding queries return two rows: manufacturers that produce a kind of shoe and the lead time for ordering the product.

```
manu_name          lead_time

Anza                  5
Hero                  4
Karsten              21
Nikolus               8
ProCycle              9
Shimara              30
```

Note that you cannot use the predicate IN for a subquery that contains a column with a TEXT or BYTE data type.

Add the keyword NOT to IN or to EXISTS to create a search condition that is the opposite of the one in the preceding queries. You also can substitute !=ALL for NOT IN.

Here are two different 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, you can use the SET EXPLAIN command to get a listing of the query plan. SET EXPLAIN is discussed in Chapter 4 of this manual and in Chapter 7 of *The Informix Guide to SQL: Reference.*

```
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)
```

Both of these statements return the following 11 rows identifying customers who have not placed orders:

```
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
```

Note that 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 later in this chapter.

The following SELECT statement identifies all the items in the **stock** table that have not been ordered yet by performing a subquery on the **items** table:

```
SELECT stock.* FROM stock
    WHERE NOT EXISTS
        (SELECT * FROM items
            WHERE stock.stock_num = items.stock_num
                AND stock.manu_code = items.manu_code)
```

It returns these rows:

```
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
```

Note that there is no logical limit to the number of subqueries a SELECT statement can have, but there is a physical limit on the size of any statement when considered as a character string. However, this limit probably is larger than any practical statement 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.*

```
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)
```

This SELECT statement 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. Assuming that no discount has been applied, such rows must have been entered incorrectly in the database. For example:

```
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 functions enable you to use SELECT statements to check the integrity of your database after you have performed an update, insert, or delete. They can be useful when you are transferring data to a history table, for example, and want to verify that the correct data is in the history table before you delete it from the original table.

Union

The union function uses the UNION keyword or operator to combine two queries into a single *compound query.* You can use the UNION keyword between two or more SELECT statements to *unite* them and produce a temporary table containing rows that exist in any or all of the original tables. (Note that you cannot use a UNION operator inside a subquery or in the definition of a view.) Figure 3-1 illustrates 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	
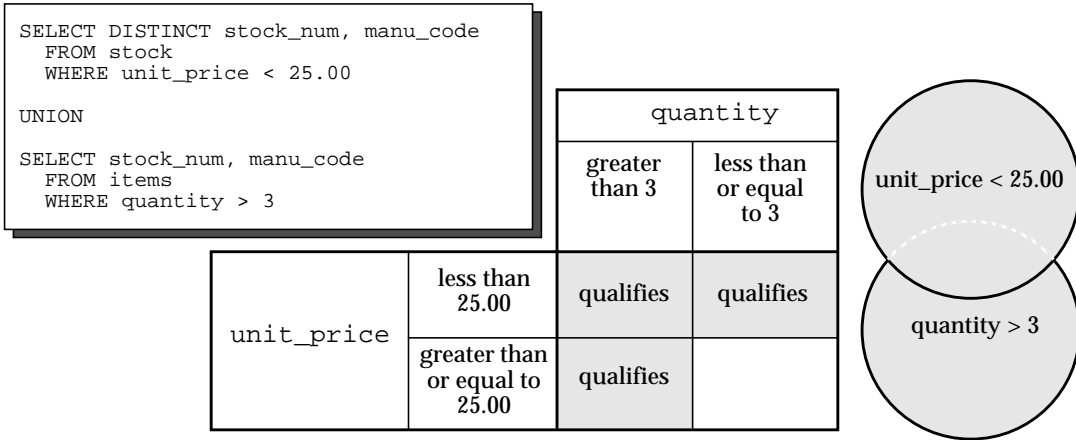
Figure 3-1   The union set operation

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 selected from each table must be of the same data type (CHARACTER type columns must be the same length), and these corresponding columns must either all allow or all disallow nulls.

This compound SELECT statement performs a union on the **stock_num** and **manu_code** columns in the **stock** and **items** tables:

```
SELECT DISTINCT stock_num, manu_code
    FROM stock
    WHERE unit_price < 25.00

UNION

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

This SELECT statement 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**:

```
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 the *last* SELECT statement and use an integer, not an identifier, to refer to the ordering column. Ordering takes place after the set operation is complete.

```
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
```

This compound query selects the same rows as the previous SELECT statement but displays them in order of manufacturer code:

```
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 to retain the duplicate values.

```
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
```

This compound SELECT statement uses the UNION ALL keywords to unite
two SELECT statements and puts the results into a temporary table by adding
an INTO TEMP clause after the final SELECT. It returns the same rows as the
preceding example but also includes duplicate values.

```
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.

```
SELECT DISTINCT state
    FROM customer
    WHERE customer_num BETWEEN 120 AND 125

UNION

SELECT DISTINCT code
    FROM state
    WHERE sname MATCHES "*a"
```

This SELECT statement selects the **state** column from the **customer** table and the corresponding **code** column from the **state** table. It returns state code abbreviations for customer numbers 120 through 125, or for states whose **sname** ends in A or a.

```
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 this example, the column name **state** from the first SELECT statement was used instead of the column name **code** from the second.

This SELECT statement performs a union on three tables. The maximum number of unions depends on the practicality of the application and any memory limitations.

```
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
```

This compound query selects items where the **unit_price** in the **stock** table is greater than $600, or the **catalog_num** in the **catalog** table is 10025, or the **quantity** in the **items** table is 10, and orders it by **manu_code**:

```
stock_num manu_code

        5 ANZ
        9 ANZ
        8 ANZ
        4 HSK
        1 HSK
      203 NKL
        5 NRG
      106 PRC
      113 SHM
```

See Chapter 7 of *The Informix Guide to SQL: Reference* for the complete syntax of the SELECT statement and the UNION operator. See also Chapters 6 and 7 in this manual, as well as the product manuals, for information specific to the **INFORMIX-4GL** and **INFORMIX-ESQL/C** products and any limitations involving the INTO clause and compound queries.

The following example 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 for ordering the retrieved rows.

```
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
```

This query pair creates a list where the California customers, the ones called most frequently, appear first:

```
sortkey   1
lname     Albertson
fname     Frank
company   Sporting Place
city      Redwood City
state     CA
phone     415-886-6677

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 of 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.
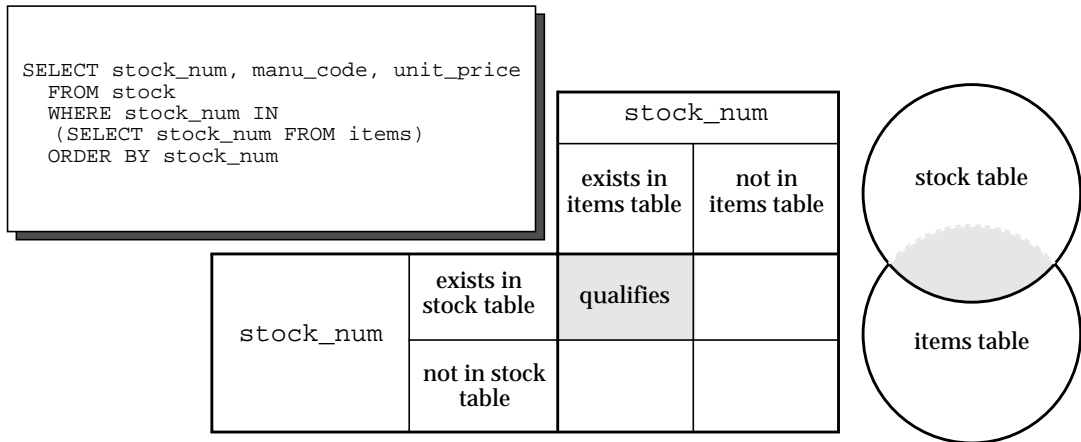
Figure 3-2   The intersection set operation

The following example of a nested SELECT statement shows the intersection of the **stock** and **items** tables:

```
SELECT stock_num, manu_code, unit_price
    FROM stock
    WHERE stock_num IN
        (SELECT stock_num FROM items)
    ORDER BY stock_num
```

The results contain all the elements from both sets, returning the following 57 rows:

```
stock_num manu_code unit_price

        1 HRO          $250.00
        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 containing 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.
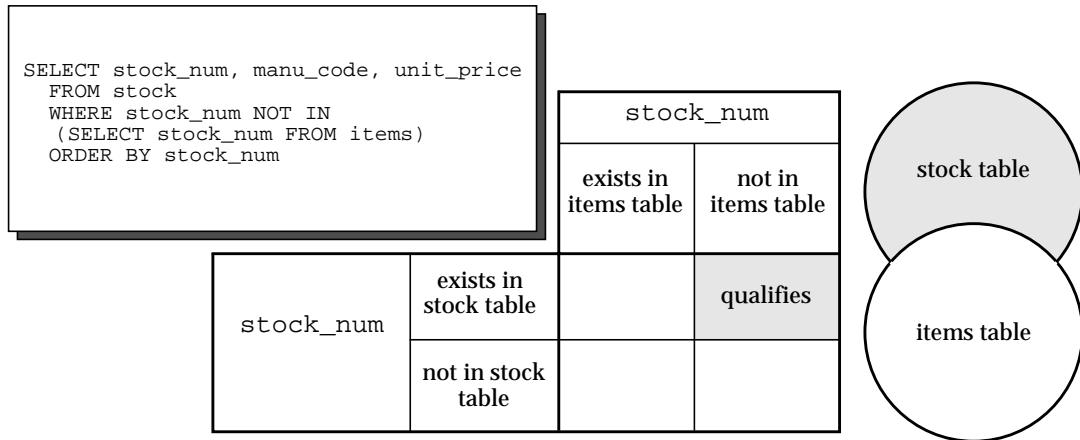
Figure 3-3   The difference set operation

The following example of a nested SELECT statement shows the difference between the **stock** and **items** tables:

```
SELECT stock_num, manu_code, unit_price
    FROM stock
    WHERE stock_num NOT IN
        (SELECT stock_num FROM items)
    ORDER BY stock_num
```

The results contain all the elements from only the first set, returning the following 17 rows:

```
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 was built on concepts introduced in Chapter 2. It provided sample syntax and results for more advanced kinds of SELECT statements, which are used to perform a query on a relational database. This chapter

- Introduced the GROUP BY and HAVING clauses, which can be used with aggregates to return groups of rows and apply conditions to those groups.

- Described how to use the rowid to retrieve internal record numbers from tables and system catalog tables, and discussed the serial internal table identifier or tabid.

- Showed how to join a table to itself with a self-join to compare values in a column to other values in the same column, and to identify duplicates.

- Introduced the keyword OUTER, explained how an outer join treats two or more tables asymmetrically, and provided examples of the four kinds of outer join.

- Described how to create correlated and uncorrelated subqueries by nesting a SELECT statement in the WHERE clause of another SELECT statement, and showed the use of aggregate functions in subqueries.

- Demonstrated the use of the keywords ALL, ANY, EXISTS, IN, and SOME in creating subqueries, and the effect of adding the keyword NOT or a relational operator.

- Discussed the set operations union, intersection, and difference.

- Showed how to use the UNION and UNION ALL keywords to create compound queries consisting of two or more SELECT statements.

Optimizing Your Queries

Chapter Overview

How much time should a query take? How many disk operations should the computer perform while executing a query? For many queries, it does not matter as long as the computer finds the information faster than a human can. But some queries must be performed in a limited amount of time, or must use only a limited amount of machine power.

This chapter reviews techniques for making queries more efficient. *It assumes that you work with an existing database* and cannot change its arrangement of tables. (Techniques of designing a new database for reliability and performance are discussed in Chapters 8 through 11 in this manual.)

This chapter covers the following topics:

- A general discussion of techniques for optimizing software, emphasizing all the things to look at before you change any SQL statement.
- A description of the *optimizer*, the part of the database server that decides how to perform a query. When you know how the optimizer forms a query plan, you can help it form more efficient ones.
- A discussion of the operations that take time during a query, so you can better choose between fast and slow operations.
- An assortment of techniques to help the optimizer choose the fastest way to accomplish a query.

This chapter concentrates on SQL performance, but performance problems can arise from other parts of the programs in which SQL statements are embedded. Two books that address general issues of performance are *The Elements of Programming Style* by Kernighan and Ritchie (McGraw-Hill 1978) and *Writing Efficient Programs* by Jon Louis Bentley (Prentice-Hall 1982). Much of Bentley's advice can be applied to SQL statements and database design.

Optimizing Techniques

Before you begin optimizing your SQL statements, begin considering them as part of a larger system that includes these components:

- One or more *programs*

  The saved queries, compiled screen forms and reports, and programs in one or more languages in which SQL statements are embedded.

- One or more *stored procedures.*

  The compiled procedures, made up of SQL and SPL (Structured Procedure Language) statements, which are stored in an executable form in the database.

- One or more *computers*

  The machines that store the programs and the databases.

- One or more *maintainers*

  The people responsible for maintaining the programs.

- One or more *users*

  The people whose work the system is supposed to amplify or simplify.

- One or more *organizations*

  The groups that own the computers and choose what work is to be done.

You may work in a large corporation where each of these components is separate. Or you may be the proprietor, maintainer, and sole user of a desktop workstation. In every case, it is important to recognize two points about the system as a whole. First, the goal of the system is to serve its *users.* Any effort you spend optimizing programs is wasted unless the result helps the users in some way. Second, SQL statements are only a small part of the system. Often, the most effective improvements can be found in other parts of the system.

The following paragraphs outline a general procedure for analyzing any computer performance problem. Follow the procedure to help avoid overlooking possible solutions, including the nontechnical ones that sometimes provide the best answer to performance problems.

Verifying the Problem

Before you begin optimizing SQL statements, be sure the problem lies with the program. How much more effective will the user be if you make the program faster? If the answer is "not much," look elsewhere for a solution.

Considering the Total System

Consider the entire system of programs, computers, and users within an organization. You might find a better solution by changing schedules or managing resources differently. Maybe the problem operation would be faster if it were done at a different time, or on a different machine, or with different jobs running at the same time. Or maybe not. But you need to consider these possibilities before you decide your best course of action.

Understanding the Application

Learn all about the application as a whole. A database application usually has many parts, such as saved queries, screen forms, report specifications, and programs. Consider them together with the user's procedures, and ask yourself the following questions:

- *What* is being done?

  You may identify excess or redundant steps that can be eliminated. Even if every step or action is important, it will help you to know them all and their sequence of use.

- *Why* is it being done?

  Especially in older applications, you may find steps that serve no purpose; for example, an operation might have been put in long ago for debugging and never removed.

- *For whom* is it being done?

  Make sure that all output from the application is wanted and used by someone. You may find output that nobody wants any more. Or the output may be needed infrequently or in a simpler format.

- *Where* is the slow part?

  Isolate as closely as possible which steps are too slow. Your time is limited, too; you want to spend time where it yields the best results.

By taking the time to understand the application, you can know all of its parts, how they are used, whether they are all essential, and which parts are too slow.

Measuring the Application

You cannot make meaningful progress on a performance problem until you have measured it. You must find a way to take repeatable, quantitative measurements of the slow parts of the application. This is crucial for the following reasons:

- Without numbers you cannot accurately and specifically describe the problem to users or to the organization.

  To convey the problem appropriately, you need to determine measurements such as "The report runs in 2 hours 38 minutes," or "The average update takes 13 seconds, but during peak hours it takes up to 49 seconds."

- Without numbers you cannot set meaningful goals.

  When you have specific measurements, you can obtain agreement on numeric goals for performance.

- Without numbers you cannot measure and demonstrate your progress.

  You need measurements to detect and demonstrate small improvements, and to choose between alternative solutions.

Manual Timing

You can obtain repeatable measurements with a hand-operated stopwatch. With practice, most people can obtain timings that are repeatable within two-tenths of a second. Manual timing is useful if you are measuring only a few events that are at least several seconds long.

Time from Operating System Commands

Your operating system probably has a command that displays the time. You can package an operation to be timed between two time commands in a command script.

Time from the Programming Language

Most programming languages have a library function for the time of day. If you have access to the source code, you can insert statements to measure the time of specific actions. For example, if the application is written in **INFOR**-

MIX-**4GL**, you can use the CURRENT function to obtain the current time as a DATETIME value. A **4GL** program can perform automated timing with code similar to the following fragment:

```
DEFINE start_time DATETIME HOUR TO FRACTION(2),
       elapsed INTERVAL MINUTE(4) TO FRACTION(2)
LET start_time = EXTEND(CURRENT,HOUR TO FRACTION(2))

       { -- here perform the operation to be timed -- }

LET elapsed = EXTEND(CURRENT,HOUR TO FRACTION(2))-start_time
DISPLAY "Elapsed time was ",elapsed
```

Elapsed time, in a multiprogramming system or network environment, does not always correspond to execution time. Most C libraries contain a function that returns the CPU time of a program, and C functions can be called from **4GL** programs and **ACE** reports.

Finding the Guilty Functions

In most programs, a very small fraction of the code (typically 20% or less) accounts for the bulk of the program execution time (typically 80% or more). The *80-20 rule*, as it is called, holds in most cases; often, the proportions are even greater. After you establish a timing mechanism, use it to define the *hot spots* in the application, that is, the statements that consume most of the time.

Keeping an Open Mind

With embedded SQL, in which one statement can trigger thousands of disk accesses, it is likely that the guilty 20% includes some SQL. However, *this is by no means certain*. If you do not begin your examination of the problem with an open mind, it is easy to overlook the part of a program that is consuming the most time.

If the sluggish operations are not due to the SQL, turn to the books cited on page 4-3. If the SQL statements need attention, you need to understand the actions of the query optimizer.

The Query Optimizer

The *optimizer* component of the database server decides how to perform a query. Its most important job is to decide the order in which to examine each table row. To make that decision, it has to decide on the most efficient way to access each table—by a sequential scan of its rows, by an existing index, or by a temporary index built for the occasion—and it has to estimate how many rows each table contributes to the final result.

Optimizer design is not a science, and the optimizer is one part of a database server that is under continuous development and improvement. This discussion reflects the optimizer used in Version 5.0 of Informix database servers. Informix database servers with earlier version numbers have an older, simpler optimizer.

The Importance of Table Order

The order in which tables are examined has an enormous effect on the speed of a join operation. This concept can be clarified with some examples that show how the database server works.

A Join Without Filters

Here is a SELECT statement that calls for a three-way join:

```
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
    GROUP BY C.customer_num, O.order_num
```

For the moment, imagine that indexes were never invented. Without indexes, the database server has no choice but to perform this operation using a simple nested loop. One of two practical *query plans* is displayed in Figure 4-1,

expressed in a programming pseudocode. A query plan states the order in which the database server examines tables, and the methods by which it accesses the tables.

```
for each row in the customer table do:
    read the row into C
    for each row in the orders table do:
        read the row into O
        if O.customer_num = C.customer_num then
            let Sum = 0
            for each row in the items table do:
                read the row into I
                if I.order_num = O.order_num then
                    let Sum = Sum + I.total_price
                end if
            end for
            prepare an output row from C,I,and Sum
        end if
    end for
end for
```

Figure 4-1 **A query plan in pseudocode**

This procedure reads the following rows:

- All rows of the **customer** table once

- All rows of the **orders** table once for each row of the **customer** table

- All rows of the **items** table once for each row of **orders** that appears in the output (every row of **orders** should appear once)

This is not the only possible query plan; another merely reverses the roles of **customer** and **orders**: for each row of **orders**, it reads all rows of **customer** looking for a matching **customer_num**. It reads the same number of rows, although in a different sequence.

The number of rows in each table is kept in the system catalog table named **systables**. You can write the following query to calculate the total count of rows read by the query plan in Figure 4-1:

```
SELECT C.nrows + C.nrows*O.nrows + O.nrows*I.nrows
    FROM systables C, systables O, systables I
    WHERE C.tabname = "customer"
        AND O.tabname = "orders"
        AND I.tabname = "items"
```

That is essentially how the optimizer predicts the amount of work required by this query plan. (The row counts in **systables** are only updated when the UPDATE STATISTICS command is run. Thus, the optimizer sometimes works from outdated information.)

A Join with Column Filters

In the preceding example, there was no difference in the amount of work that would be done by the two possible query plans. The presence of a *column filter* changes things. A column filter is a WHERE expression that reduces the number of rows that a table contributes to a join. Here is the preceding query with a filter added:

```
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 expression `O.paid_date IS NULL` *filters out* some rows, reducing the number of rows that are used from the **orders** table. As before, two query plans are possible. The plan that starts by reading from **orders** is displayed in pseudocode in Figure 4-2.

```
for each row in the orders table do:
    read the row into O
    if O.paid_date is null then
        for each row in the customer table do:
            read the row into C
            if O.customer_num = C.customer_num then
                let Sum = 0
                for each row in the items table do:
                    read the row into I
                    if I.order_num = O.order_num then
                        let Sum = Sum + I.total_price
                    end if
                end for
                prepare an output row from C,I,and Sum
            end if
        end for
    end if
end for
```

Figure 4-2   One of two query plans in pseudocode

Let *pdnull* stand for the number of rows in **orders** that pass the filter. It is the number that the following query returns:

```
SELECT COUNT(*) FROM orders WHERE paid_date IS NULL
```

Assume that there is just one customer for every order, as there should be. Then we can say that the plan in Figure 4-2 reads these rows:

- All rows of the **orders** table once
- All rows of the **customer** table, *pdnull* times
- All rows of the **items** table, *pdnull* times

```
for each row in the customer table do:
    read the row into C
    for each row in the orders table do:
        read the row into O
        if O.paid_date is null and
           O.customer_num = C.customer_num then
            let Sum = 0
            for each row in the items table do:
                read the row into I
                if I.order_num = O.order_num then
                    let Sum = Sum + I.total_price
                end if
            end for
            prepare an output row from C,I,and Sum
        end if
    end for
end for
```

Figure 4-3 *The alternative query plan in pseudocode*

An alternative plan is shown in Figure 4-3; it reads from **customer** first. Because the filter is not applied in the first step, this plan reads these rows:

- All rows of the **customer** table once
- All rows of the **orders** table, once for every row of **customer**
- All rows of the **items** table, *pdnull* times

The query plans in Figure 4-2 and Figure 4-3 produce the same output, although in different sequences. They differ in that one reads a table *pdnull* times while the other reads a table SELECT COUNT(*) FROM customer times. The choice the optimizer makes between them could make a difference of thousands of disk accesses in a real application.

Using Indexes

The preceding examples do not use indexes or constraints. That was unrealistic; almost all tables have one or more indexes or constraints, and their presence makes a difference in the query plan. Figure 4-4 shows the outline of a query plan for the previous query as it might be constructed using indexes. (See also "The Structure of an Index" on page 4-26.)

```
for each row in the customer table do:
    read the row into C
    look up C.customer_num in index on orders.customer_num
    for each matching row in the orders table do:
        read the row into O
        if O.paid_date is null then
            let Sum = 0
            look up O.order_num in index on items.order_num
            for each matching row in the items table do:
                read the row into I
                let Sum = Sum + I.total_price
            end for
            prepare an output row from C,I,and Sum
        end if
    end for
end for
```

Figure 4-4   ***A query plan using indexes in pseudocode***

The keys in an index are sorted so that when the first matching key is found, any other rows with identical keys can be read without further searching. This query plan reads only the following rows:

- All rows of the **customer** table once
- All rows of the **orders** table once
- Those rows of the **items** table that match to *pdnull* rows from **orders**

This is a huge reduction in effort compared to the plans without indexes. (An inverse plan, reading **orders** first and looking up rows in **customer** by its index, is equally effective.)

However, any plan that uses an index must read index data from disk as well as read row data. It is difficult to predict the number of index pages that are read, since some number of recently used index pages are retained in memory and are found there when they are needed. (See "Disk-Access Management" on page 4-21.)

The physical order of a table also can affect the cost of index use. The query plan in Figure 4-4 reads the rows of the **customer** table in physical order. If that is also customer-number order, customer numbers are presented in sequence to the index on **orders.customer_num**. Since the index is sorted, the result is to read the index pages sequentially. Almost certainly, they are read only once.

The physical order of the **customer** table is customer-number order if a clustered index is on that column. It is approximately that order if customer numbers are SERIAL values generated by the database server.

However, if the physical order is random with respect to customer numbers, each search of the index on **orders** might lead to a different page of index keys, so that an index page is read for almost every row.

The Sort-Merge Join Technique

The sort-merge table join, implemented in Version 5.0, provides an alternative to the nested-loop table join. It allows the optimizer to choose the best path of execution to produce the fastest possible join. This may be a loop join, a sort-merge join, or a combination of the two.

Sort-merge functionality optimizes the execution of certain queries without degrading performance where sort merge is not used. A sort-merge join happens automatically when at least one filter of the join is an equality operator.

The sort-merge join does not change the fundamental strategy of the optimizer. Rather, it provides a richer set of alternatives to nested-loop joins using temporary indexes and alters the way the ORDER BY and GROUP BY paths are analyzed. The path chosen by the optimizer is displayed in the output when the SET EXPLAIN ON statement is issued, as described in Chapter 7 of *The Informix Guide to SQL: Reference* and in "Reading the Plan" on page 4-18.

How the Optimizer Works

The optimizer in Version 4.0 and later Informix database servers formulates all possible query plans. For each plan, it estimates the number of table rows to be examined, disk pages to be read, and (when the **INFORMIX-STAR** feature is installed) network accesses that the plan requires. It selects the plan with the lowest estimate.

Selecting an Optimization Level

With Version 5.0, you can specify a high or low level of database server optimization with the SET OPTIMIZATION statement. This statement is described in detail in Chapter 7 of *The Informix Guide to SQL: Reference.*

The default (high) optimization level is a sophisticated, cost-based strategy that examines all the reasonable choices and selects the best overall alternative. However, for large joins, this level may incur more overhead than you want; in extreme cases, performance degradation can be severe. The low optimization level eliminates unlikely join strategies during the early stages, reducing the amount of time and resources spent during optimization. The risk with this level is that the optimal strategy is not selected because it was eliminated from consideration during the early stages. You normally obtain optimum overall performance with the default optimization level. If experimentation with your application reveals that the low optimization level is best, then you should set your optimization level to low.

Providing Input

The optimizer can only be successful if its estimates are accurate. (They need not be accurate in absolute terms, but they must be relatively accurate so that better plans produce lower estimates than worse plans.) However, the optimizer has only limited information. To keep its own work down to a small fraction of the execution time, it has to make do with the information in the system catalog tables and other summary information. There is no time, for example, for the optimizer to perform a SELECT COUNT(*) operation to obtain an accurate count of the rows in a table.

The information available to the optimizer comes from the system catalog tables. (See Chapter 2 of *The Informix Guide to SQL: Reference* for information on system catalog tables.) In all Informix database servers, this information includes

- The number of rows in a table (as of the most recent UPDATE STATISTICS command)
- Whether a column is constrained to be unique
- The indexes that exist on a table, including which columns they encompass, whether they are ascending or descending, and whether they are clustered

The system catalog tables maintained by **INFORMIX**-**OnLine** supply the following additional input:

- The number of disk pages occupied by row data
- The depth of the index B+ tree structure (a measure of the amount of work needed to perform an index lookup)
- The number of disk pages occupied by index entries
- The number of unique entries in an index (divided by the number of rows, this suggests how many rows might match to a given key)
- Second-largest and second-smallest key values in an indexed column

Only the second-largest and second-smallest key values are noted because the extreme values might be special out-of-range signals. The database server assumes that key values are distributed smoothly between the second largest and second smallest. Only the initial four bytes of these keys are stored.

Assessing Filters

The optimizer first examines the expressions in the WHERE clause by looking for filters. The optimizer estimates the *selectivity* of each filter it finds. The selectivity is a number between 0 and 1 that indicates the fraction of rows the optimizer thinks the filter will pass. A very selective filter that passes very few rows is assigned a selectivity near 0; a filter that passes most rows is assigned a selectivity near 1. For details on this process, see "Filter Selectivity Assignments" on page 4-16.

The optimizer also notes other information about the query, such as whether an index is sufficient. If only an indexed column is selected, it is faster to read the index pages and not read rows at all. (This is often possible with subqueries.)

In addition, the optimizer notes whether an index can be used to evaluate a filter. For this purpose, an indexed column is a column that has an index, or one that is named first in a composite index. There are several cases to consider:

- When an indexed column is compared to a literal, a host variable, or an uncorrelated subquery, the database server can look up matching values in the index instead of reading the rows.
- When an indexed column is compared to a column in another table (a join expression), the database server can use the index to find matching

values, provided that the query plan calls for reading rows from the other table first. The following join expression is an example:

```
WHERE customer.customer_num = orders.customer_num
```

If rows of **customer** are read first, values of **customer_num** can be looked up in an index on **orders.customer_num**.

- Whether an index can be used in processing an ORDER BY clause. If all the columns in the clause appear in one index in the same sequence, the

Filter Selectivity Assignments

The following table lists some of the selectivities that the optimizer assigns to filters of different types. It is not an exhaustive list, and other expression forms may be added in the future.

Expression Form	Selectivity (F)
indexed-col = literal-value	
indexed-col = host-variable	
indexed-col IS NULL	$F = 1/$(number of distinct keys in index)
tab1.indexed-col = tab2.indexed-col	$F = 1/$(number of distinct keys in the larger index)
indexed-col > literal-value	$F = (\textit{2nd-max - literal-value})/(\textit{2nd-max - 2nd-min})$
indexed-col < literal-value	$F = (\textit{literal-value - 2nd-min})/(\textit{2nd-max - 2nd-min})$
any-col IS NULL	
any-col = any-expression	$F = 1/10$
any-col > any-expression	
any-col < any-expression	$F = 1/3$
any-col MATCHES *any-expression*	
any-col LIKE *any-expression*	$F = 1/5$
EXISTS *subquery*	$F = 1$ if *subquery* estimated to return >0 rows, else 0
NOT *expression*	$F = 1 - \text{F}(\textit{expression})$
expr1 AND *expr2*	$F = \text{F}(\textit{expr1}) \times \text{F}(\textit{expr2})$
expr1 OR *expr2*	$F = \text{F}(\textit{expr1}) + \text{F}(\textit{expr2}) - \text{F}(\textit{expr1}) \times \text{F}(\textit{expr2})$
any-col IN *list*	treated as $\textit{anycol} = \textit{item}_1$ OR…OR $\textit{anycol} = \textit{item}_n$
any-col relop ANY *subquery*	treated as *any-col relop* $\textit{value}_1$ OR…OR *anycol relop* $\textit{value}_n$ for estimated size of subquery *n*

Key:
indexed-col: first or only column in an index (**INFORMIX-OnLine** only)
2nd-max, 2nd-min: second-largest and -smallest key values in indexed column (**INFORMIX-OnLine** only)
any-col: any column not covered by a preceding formula

database server can use the index to read the rows in their ordered sequence, thus avoiding a sort.

• Whether an index can be used in processing a GROUP BY clause. If all the columns in the clause appear in one index in the same sequence, the database server can read groups with equal keys from the index without needing a sort.

Selecting Table-Access Paths

The optimizer next chooses what it estimates to be the most efficient way of accessing each table named in the query. It has four choices:

• To read the rows of the table sequentially

• To read one of the indexes for the table and read the rows to which the index points

• To create and use a temporary index

• To perform a sort merge

The choice between the first two options depends in large part on the presence of a filter expression. When there is any filter on an indexed column, the database server chooses the selected rows through the index and processes only those rows.

When there is no filter, the database server must read all of the rows anyway. It is usually faster simply to read the rows, rather than reading the pages of an index and then reading the rows. However, if the rows are required in sorted order, there may be a net savings in reading the index and through it reading the rows in sorted order.

The optimizer might choose a third option, creating a temporary index, in two cases. When neither table in a join has an index on the joining column, and the tables are large enough, the optimizer may decide that it is quicker to build an index to one table than to read through that table sequentially for each row of the other table. You can also use a temporary index to generate rows in sorted or grouped sequence. (The alternative is to write the output rows to a temporary table and sort that.)

The optimizer has a fourth option—performing a sort merge—which does not require a temporary index. This happens automatically when at least one filter of the join is an equality operator.

Selecting the Query Plan

With all this information available, the optimizer generates all possible query plans for joining tables in pairs. Then, if more than two tables are joined, the optimizer uses the best two-table plans to form all plans for joining two tables to a third, three to a fourth, and so on.

The optimizer adds any final work to the completed join plans. For example, if there is an ORDER BY or GROUP BY clause, and a plan does not produce rows in ordered sequence, the optimizer adds to the plan an estimate of the cost of sorting the output. Sorting is discussed in "Time Cost of a Sort" on page 4-21.

Finally, the optimizer selects a plan that appears to promise the least amount of work, and passes it to the main part of the database server to be executed.

Reading the Plan

The choice the optimizer makes does not have to remain a mystery; you can determine exactly what query plan it chooses. Execute the statement SET EXPLAIN ON before you execute a query. Beginning with the next query, the optimizer writes an explanation of its query plan to a particular file (the name of the file and its location depend on the operating system in use). A typical explanation is shown in Figure 4-5.

After repeating the query, the optimizer shows its estimate of the work to be done (104 in Figure 4-5) in arbitrary units. A single disk access is one unit, and other actions are scaled to that. This query plan was chosen over others because its estimate was the lowest.

```
QUERY:
------
SELECT C.customer_num, O.order_num, SUM (I.total_price)
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
        AND O.order_num = I.order_num
    GROUP BY C.customer_num, O.order_num;

Estimated Cost: 104
Estimated # of Rows Returned: 2

1) pubs.o: INDEX PATH

 (1) Index Keys: order_num

2) pubs.c: INDEX PATH

 (1) Index Keys: customer_num (Key-Only)
 Lower Index Filter: pubs.c.customer_num = pubs.o.customer_num

3) pubs.i: INDEX PATH

 (1) Index Keys: order_num
 Lower Index Filter: pubs.i.order_num = pubs.o.order_num
```

Figure 4-5   ***Typical output produced by the optimizer with SET EXPLAIN ON***

The optimizer also reveals its estimate of the number of rows the query pro-
duces. In Figure 4-5, the optimizer incorrectly estimates 2. The estimate is
incorrect because the optimizer has no good way to tell how many groups the
GROUP BY clause produces. You know that there is a group for every row in
the **orders** table; the optimizer cannot know that.

In the body of the explanation, the optimizer lists the order in which tables
are accessed and the method, or access path, by which it reads each table. The
following breakdown clarifies the plan:

1. The **orders** table is read first. The index on **order_num** is used; hence the
    rows are read in **order_num** sequence.

    Since all rows are read, it actually is less effort to read the table sequen-
    tially. However, retrieving the rows in **order_num** sequence lets the
    GROUP BY clause be performed without a final sort.

2. For each row of **orders**, a search is made for matching rows in the **customer** table. The search uses the index on **customer_num**.

   The notation *Key-Only* means that since only the **customer_num** column is used in the output, only the index is read; no row is read from the table.

3. For each row of **orders** that has a matching **customer_num**, a search is made in the **items** table using the index on **order_num**.

It is not always obvious why the optimizer makes its choices. By comparing the plans produced by several variations in the query, you can usually deduce some of its logic. However, it may be best to read the explanation file accepting that this is what the optimizer *does* without trying too hard to see *why* it chooses to do it.

Time Costs of a Query

To execute a query, the database server spends most of its time in performing two types of operation: reading data from disk, and comparing column values. Of the two, reading data is, by far, the slower task. This section examines where the database server spends time. The next section explores the implications of making queries faster.

Activities in Memory

The database server can only process data in memory. It must read a row into memory before it can test it with a filter expression. It must read rows from both tables before it can test a join condition. The database server prepares an output row in memory by assembling the selected columns from other rows in memory.

Most of these activities go very quickly. Depending on the computer, the database server can perform hundreds or even thousands of comparisons a second. As a result, the time spent on in-memory work is usually a small part of the whole execution time.

Two in-memory activities can take a significant amount of time. One is sorting, as described in "Time Cost of a Sort" on page 4-21. The other activity is processing comparisons using LIKE and MATCHES, especially those that test for "zero or more" characters at the front or in the middle of values.

Disk-Access Management

It takes much longer to read a row from disk than to examine a row in memory. The main goal of the optimizer is to reduce the amount of data that must be read from disk, but it can eliminate only the most obvious inefficiencies.

Time Cost of a Sort

A sort requires both in-memory work and disk work.

The in-memory work is proportional to $c*w*n*\log_2(n)$, where

c is the number of columns being ordered, and represents the costs of extracting column values from the row and concatenating them into a sort key.

w is proportional to the width of the combined sort key in bytes, and stands for the work of copying or comparing one sort key. A numeric value for w would depend strongly on the computer hardware in use.

$n*\log_2(n)$ is the number of comparisons that are made while sorting a table of n rows.

The disk work is proportional to $2*n*m$, where n is again the number of rows. The factor m represents the number of *levels of merge* the sort must use, a factor that depends on the number of sort keys that can be held in memory.

When all the keys can be held in memory, $m=1$ and the disk work is proportional to $2n$. In other words, the rows are read, sorted in memory, and written. (When building an index, only index pages are written.)

For tables from moderate to large sizes, rows are sorted in batches that fit in memory and then the batches are merged. When $m=2$, the rows are read, sorted, and written in batches. Then the batches are read again and merged and written again, resulting in disk work proportional to $4n$. For extremely large tables, the batches must be broken into batches. The disk work is then proportional to $6n$. In short, as table size goes up there are sudden, discontinuous changes in the amount of disk work in a sort, from $2n$ to $4n$ to $6n$. The table size at which these steps occur depends on many factors, only one of which (key size) is in your control.

The best way to reduce the cost of sorting is to find a way to sort fewer rows, since the factor n dominates both expressions. When that is not possible, look for ways to sort on fewer and narrower columns. Not only does this reduce the factors c and w, it defers the step to the next merge level.

Disk Pages

The database server deals with disk storage in units called *pages*. A page is a block of fixed size. The same size is used for all databases managed by one database server. Indexes are also stored in page-size units.

The size of a page depends on the database server. With **INFORMIX-OnLine**, the page size is set when **OnLine** is initialized. It is usually 2 kilobytes (2,048 bytes), but you can ask the person who installed **OnLine** to tell you what was chosen. Other Informix database servers use the file storage of the host operating system, so their page size is the block size used by the host operating system. One kilobyte (1,024 bytes) is a typical size, but you should check it for the operating system you use.

It is possible to define tables so wide that one row fills a page (some database servers permit a row to exceed the size of a page). However, a typical row size is between 50 and 200 bytes, so in a typical table a disk page contains from 5 to 50 rows. An index entry consists of a key value and a 4-byte pointer, so an index page typically contains from 50 to 500 entries.

Page Buffers

The database server has a set of memory spaces in which it keeps copies of the disk pages it read most recently. It does this in the hope that these pages will be needed again. If they are, the database server will not have to read them from disk.

Like the size of a disk page, the number of these page buffers depends on the database server and the host operating system.

The Cost of Reading a Row

When the database server needs to examine a row that is not already in memory, it must read it from disk. It does not read just one row; it reads the pages that contain the row. (When a row is larger than a page, it reads as many whole pages as necessary.) The cost of reading one page is the basic unit of work that the optimizer uses for its calculations.

The actual cost of reading a page is variable and hard to predict. It is a combination of the following factors:

Buffering   The needed page might be in a page buffer already, in which case the cost of access is near zero.

Contention   If more than one application is contending for the use of the disk hardware, the database server request can be delayed.

Seek time   The slowest thing a disk does is to *seek*; that is, to move the access arm to the track holding the data. Seek time depends on the speed of the disk and the location of the disk arm when the operation starts. Seek time varies from zero to a large fraction of a second.

Latency   The transfer cannot start until the beginning of the page rotates under the access arm. This *latency*, or rotational delay, depends on the speed of the disk and on the position of the disk when the operation starts. Latency can vary from zero to a few milliseconds.

The time cost of reading a page can vary from microseconds (for a page in a buffer), to a few milliseconds (when contention is zero and the disk arm is already in position), to hundreds of milliseconds.

The Cost of Sequential Access

Disk costs are lowest when the database server reads the rows of a table in physical order. When the first row on that page is requested, its disk page is read. Requests for subsequent rows are satisfied from the buffer until the entire page is used. As a result, each page is read only once.

Provided that the database server is the only program using the disk, the seek-time cost is also minimized. The disk pages of consecutive rows are usually in consecutive locations on the disk, so the access arm moves very little from one page to the next. Even when that is not the case, groups of consecutive pages are usually located together so that only a few long seeks are needed. For example, **INFORMIX**-**OnLine** allocates disk space to a table in multipage *extents*. Extents may be separated on the disk, but within an extent the pages are close together.

Even latency costs may be lowest when pages are read sequentially. This depends on the hardware and, when operating system files are used, on the methods of the operating system. Disks are usually set up so that when pages are read sequentially, latency is minimized. On the other hand, it is possible to set up a disk so that a full rotation occurs between each sequential page, drastically slowing sequential access.

The Cost of Nonsequential Access

Disk costs are higher when the rows of a table are called for in a sequence that is unrelated to physical order. Practical tables are normally much larger than the database server page buffers, so only a small part of the table pages can be held in memory. When a table is read in nonsequential order, only a few rows are found in buffered pages. Usually one disk page is read for every row that is requested.

Since the pages are not taken sequentially from the disk, there is usually both a seek delay and a rotational delay before each page can be read. In short, when a table is read nonsequentially, the disk access time is much higher than when it is read sequentially.

The Cost of Rowid Access

The simplest form of nonsequential access is to select a row based on its *rowid* value. (The use of rowid in a SELECT statement is discussed in Chapter 3. The use of rowid in database design is discussed in Chapter 10.) A rowid value specifies the physical location of the row and its page. The database server simply reads the page, incurring the costs already noted.

The Cost of Indexed Access

There is an additional cost associated with finding a row through an index: The index itself is stored on disk, and its pages must be read into memory.

The database server uses indexes in two ways. One way is to look up a row given a key value. This is the kind of look up used when you join two tables, as in a statement such as this:

```
SELECT company, order_num
    FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
```

One table, probably **customer**, is read sequentially. Its value of **customer_num** is used to search the index on the **customer_num** column of **orders**. When a match is found, that row of **orders** is read.

An index look up works downward from the root page to a leaf page. (See "The Structure of an Index" on page 4-26.) The root page, since it is used so often, is almost always found in a page buffer. The odds of finding a leaf page in a buffer depend on the size of the index; the odds become poorer as the size of the table increases.

If a table is very large, so that the number of index leaf pages is much larger than the buffer space, almost every search causes a leaf page to be read in addition to the page containing the row. The number of pages read is approximately $2r$, where r is the number of rows to be looked up. Although it is costly, this kind of access is still a bargain, since the alternative is to perform each look up by reading the entire **orders** table, with disk costs proportional to r^2.

The other way the database server uses an index is to read it sequentially. It does this to fetch the rows of a table in a specific order other than the physical order. For this kind of access, the database server reads the leaf pages in sequence, treating them as a list of rows in key sequence. Since the index pages are read only once, the total of disk operations is proportional to $r+i$, where i is the number of leaf pages in the index.

The Cost of Small Tables

One conclusion you can draw from the preceding paragraphs is that small tables are never slow. A table is "small" if it occupies so few pages that it can be retained entirely in the page buffers. Any table that fits in four or fewer pages is certainly "small" in this sense.

In the **stores5** database, the **state** table that relates abbreviations to names of states has a total size less than 1,000 bytes; it fits in, at most, two pages. It can be included in any query at almost no cost. No matter how it is used—even if it is read sequentially many times—it costs two disk accesses at most.

The Cost of Network Access

Whenever data is moved over a network, additional delays are imposed. Networks are used in two contexts:

- Using **INFORMIX-NET**, the application sends a query across the network to a database server in another machine. The database server performs the query using its locally attached disk. The output rows are returned over the network to the application.

- Using the **INFORMIX-STAR** (distributed data) component of **INFORMIX-OnLine**, a database server in one machine can read and update rows from tables in databases located on other machines.

Both contexts may apply; it is possible to use **INFORMIX-NET** to call on one **INFORMIX-OnLine** server, and send that server a query that causes it to use tables at yet other servers.

The data sent over a network consists of command messages and buffer-sized blocks of row data. While many differences of detail exist between the two contexts, they can be treated identically under a simple model in which one machine, the *sender*, sends a request to another, the *responder*, which responds with a block of data from a table.

The Structure of an Index

An index is arranged as a hierarchy of pages (technically, a *B+ tree*), as depicted in the figure below. At the bottom level, a sequence of *leaf pages* contains the index key values of the table rows, each with a pointer to the location of the row containing that value. The keys in the leaf pages are in key sequence.

At each higher level, a page lists the highest key value to be found in each page at the lower next level. The topmost level is a single *root page* from which any other page of the index can be found.

root page

. . .

leaf page

leaf page

. . .   . . .

. . .   . . .
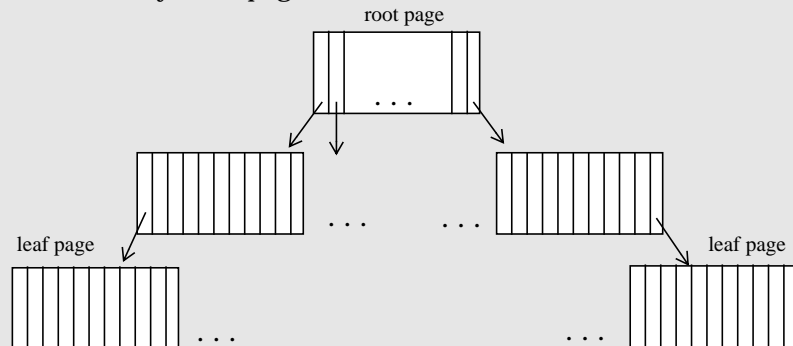
The size of one entry is the width of the indexed columns in bytes, plus 4. If you know that and the page size, you can estimate the number of entries per page. This estimate is approximate since first, some data compression is used in storing keys and second, not all pages are full.

The number of levels depends on the number of rows to be indexed and the number of entries per page. Suppose a page holds 100 keys. Then a table of 100 rows has a single index page, the root page. A table with between 101 and 10,000 rows has a two-level index consisting of a root page and from 2 to 100 leaf pages. A table with as many as 1,000,000 rows has only a 3-level index (so the preceding index must belong to a table of considerable size).

Whenever data is exchanged over a network, delays are inevitable in the following situations:

- If the network is busy, the sender must wait its turn to transmit. Such delays are typically brief, less than a millisecond. But in a heavily loaded network, they can increase exponentially to tenths of seconds and more.

- The responder may be handling requests from more than one sender, so when the request arrives, it may be queued for a time that can range from milliseconds to seconds.

- When the responder acts on the request, it incurs the time costs of disk access and in-memory operations as described in the preceding topics.

- Transmission of the response is again subject to network delays.

The important point about network access is its extreme variability. In the best case, when neither the network nor the responder is busy, transmission and queueing delays are insignificant and the responder sends a row almost as quickly as a local database server could do it. Furthermore, when the sender asks for a second row, the page is likely still to be in the responder page buffers.

Unfortunately, as network load goes up, all of these factors tend to go bad at the same time. Transmission delays rise in both directions. The queue at the responder gets longer. And the odds of a page remaining in the responder buffer become worse. Thus, network access costs can change quite suddenly from very low to extremely high.

The optimizer that **INFORMIX**-**OnLine** uses assumes that access to a row over the network takes longer than access to a row in a local database. The optimizer has no way to account for varying network loads, so it is too pessimistic at times and too optimistic at others.

Making Queries Faster

In general, you can speed up a query by changing it so that it

- Reads fewer rows
- Avoids a sort, sorts fewer rows, or sorts on a simpler key
- Reads rows sequentially instead of nonsequentially

The way to achieve these ends is not always obvious. The specific methods depend on the details of the application and the database design. The following paragraphs suggest a general approach, followed by some techniques that apply in limited circumstances.

Preparing a Test Environment

First, select a single query that is too slow. Then, set up an environment in which you can take predictable, repeatable timings of that query. Without this environment, you can never be sure whether a change helped.

If you are using a multiuser system or a network, so that system load varies widely from hour to hour, you may need to perform your experiments at the same time each day to obtain repeatable results. You may even need to work at night.

If the query presently is embedded in a complicated program, consider extracting the SELECT statement and executing it interactively, or embedding it in a simpler program.

If the real query takes many minutes or hours to complete, it may be a good idea to prepare a scaled-down database in which you can run tests more quickly. This can be helpful, but you must be aware of two potential problems:

- The optimizer can make different choices in a small database than in a large one, even when the relative sizes of tables are the same. Verify that the query plan is the same in the real and model databases.

- Execution time is rarely a linear function of table size. Sorting time, for example, increases faster than table size, as does the cost of indexed access when an index goes from two to three levels. What appears to be a big improvement in the scaled-down environment can be insignificant when applied to the full database.

Therefore, any conclusion you reach as a result of tests in the model database must be tentative until verified in the large one.

Studying the Data Model

Study the definitions of all the tables, views, and indexes used in the database. You can examine these details interactively using the Table option of **DB-Access** or **INFORMIX-SQL**. Pay particular attention to the indexes that exist, to the data types of columns used in join conditions and for ordering, and to the presence of views. This chapter is written assuming that you cannot change the data model. Nevertheless, the better you understand it, the better you can understand the query plans chosen by the optimizer.

For information on data types and views, see Chapters 9 and 11 of this manual, respectively.

Studying the Query Plan

Use SET EXPLAIN ON to determine the query plan being used. Here are some things to look for in particular:

- Unexpected tables are joined. A query against a view often joins more tables than you might expect.

- Mention of *temporary files* means the output is being written to a temporary table, after which it is sorted. The number of disk accesses is at least twice what it is if a sort were not needed.

- Use of a sequential access path for the second or subsequent table in a join means that the table is read in its entirety for every selected row of every table that precedes it in the plan.

- Use of an *autoindex* path means the database server takes the time to construct an index as a way of avoiding multiple sequential accesses.

- Use of a sequential access path for the first table in a join can be wasteful when only a small fraction of its rows appear in the output.

Rethinking the Query

Now that you understand what the query is doing, look for ways to obtain the same output with less effort. The following suggestions correspond to the preceding list.

Rewriting Joins Through Views

You might find that the query joins a table to a view that is itself a join. If you rewrite the query to join directly to fewer tables, you might be able to produce a simpler query plan.

Avoiding or Simplifying Sorts

A sort is not necessarily negative. The sort algorithm of the database server is highly tuned and extremely efficient. It is certainly as fast as any external sort program you might apply to the same data. As long as the sort is performed only occasionally, or to a relatively small number of output rows, there is no need to avoid it.

However, you should avoid or simplify repeated sorts of large tables. The optimizer avoids a sort step whenever it can produce the output in its proper order automatically by using an index. Here are some factors that prevent the optimizer from using an index:

- One or more of the ordered columns is not included in the index.
- The columns are named in a different sequence in the index and the ORDER BY or GROUP BY clause.
- The ordered columns are taken from different tables.

Another way to avoid sorts is discussed in "Using a Temporary Table to Speed Queries" on page 4-34.

If a sort is necessary, look for ways to simplify it. As discussed in "Time Cost of a Sort" on page 4-21, the sort is quicker if you can sort on fewer or narrower columns.

Eliminating Sequential Access to Large Tables

Sequential access to a table other than the very first one in the plan is ominous; it threatens to read every row of the table once for every row selected from the preceding tables. You should be able to judge how many times that is: perhaps only a few, but perhaps hundreds or even thousands.

If the table is a small one, it can do no harm to read it over and over; the table resides completely in memory. Sequential search of an in-memory table can be faster than searching the same table through an index, especially if also having its index pages in memory pushes other useful pages out of the buffers.

When the table is larger than a few pages, however, repeated sequential access is deadly to performance. One way to prevent it is to provide an index to the column being used to join the table.

The subject of indexing is discussed in the context of database design in "Managing Indexes" on page 10-19. However, any user with Resource privileges can build additional indexes. Use the CREATE INDEX command to make an index.

An index consumes disk space proportional to the width of the key values and the number of rows. (See "The Structure of an Index" on page 4-26.) Also, the database server must update the index whenever rows are inserted, deleted, or updated; this slows these operations. If necessary, you can use DROP INDEX to release the index after a series of queries, thus freeing space and making table updates easier.

Using Unions to Avoid Sequential Access

Certain forms of WHERE clauses force the optimizer to use sequential access even when indexes exist on all the tested columns. The following query forces sequential access to the **orders** table:

```
SELECT * FROM orders
    WHERE (customer_num = 104 AND order_num > 1001)
        OR order_num = 1008
```

The key element is that two (or more) separate sets of rows are retrieved. The sets are defined by relational expressions that are connected by OR. In the example, one set is selected by this test:

```
(customer_num = 104 AND order_num > 1001)
```

The other set is selected by this test:

```
order_num = 1008
```

The optimizer uses a sequential access path even though there are indexes on the **customer_num** and **order_num** columns.

You can speed queries of this form by converting them into UNION queries. Write a separate SELECT statement for each set of rows and connect them with the UNION keyword. Here is the preceding example rewritten this way:

```
SELECT * FROM orders
    WHERE (customer_num = 104 AND order_num > 1001)

UNION

SELECT * FROM orders
    WHERE order_num = 1008
```

The optimizer uses an index path for each query.

Replacing Autoindexes with Indexes

If the query plan includes an *autoindex* path to a large table, take it as a recommendation from the optimizer that an index should be on that column. It is reasonable to let the database server build and discard an index if you perform the query only occasionally, but if the query is done even daily, you save time by creating a permanent index.

Using Composite Indexes

The optimizer can use a composite index (one that covers more than one column) in several ways. In addition to its usual function of ensuring that the values in columns *abc* are unique, an index on the columns *a*, *b*, and *c* (in that order) can be used in the following ways:

- To evaluate filter expressions on column *a*
- To join column *a* to another table
- To implement ORDER BY or GROUP BY on columns *a*, *ab*, or *abc* (but not on *b*, *c*, *ac*, or *bc*)

If your application is performing several long queries, each of which involves a sort on the same columns, you might save time by creating a composite index on those columns. In effect, you perform the sort once and save its output for use in every query.

Using *tbcheck* on Suspect Indexes

With some database servers, it is possible for an index to become ineffective because it has been internally corrupted. If a query that uses an index has slowed down inexplicably, use the **bcheck** utility to check the integrity of the index and to repair it if necessary. (The **bcheck** utility does the same job for **INFORMIX-SE**.)

Dropping and Rebuilding Indexes After Updates

After extensive amounts of updating (after the replacement of a fourth or more of the rows of a table), the structure of an index can become inefficient. If an index seems to be less effective than it should be, yet **bcheck** reports no errors, try dropping the index and re-creating it.

Avoiding Correlated Subqueries

A correlated subquery is one in which a column label appears in both the select list of the main query and the WHERE clause of the subquery. Since the result of the subquery might be different for each row that the database server examines, the subquery is executed anew for every row if the current correlation values are different from the previous ones. The optimizer tries to use an index on the correlation values to cluster identical values together. This procedure can be extremely time consuming. Unfortunately, some queries cannot be stated in SQL without the use of a correlated subquery.

When you see a subquery in a time-consuming SELECT statement, look to see if it is correlated. (An uncorrelated subquery, one in which no row values from the main query are tested within the subquery, is only executed once.) If so, try to rewrite the query to avoid it. If you cannot, look for ways to reduce the number of rows that are examined; for instance, try adding other filter expressions to the WHERE clause, or try selecting a subset of rows into a temporary table and searching only them.

Avoiding Difficult Regular Expressions

The MATCHES and LIKE keywords support *wildcard* matches—technically known as *regular expressions.* Some regular expressions are more difficult than others for the database server to process. A wildcard in the initial position, as in the following example (find customers whose first names do not end in *y*), forces the database server to examine every value in the column:

```
SELECT * FROM customer WHERE fname NOT LIKE "%y"
```

The optimizer does not attempt to use an index for such a filter, even when one exists. It forces sequential access for the table. If the table is the second or later one in a join, the query is slow.

If a difficult test for a regular expression is essential, avoid combining it with a join. First process the single table, applying the test for a regular expression to select the desired rows. Save the result in a temporary table, and join that table to the others.

Regular-expression tests with wildcards only in the middle or at the end of the operand do not prevent the use of an index when one exists. However, they still can be slow to execute. Depending on the data in the column, you can convert some expressions to range tests using relational operators. The following SELECT statement is an example:

```
SELECT * FROM customer WHERE zipcode LIKE "98_ _ _"
```

You can rewrite that SELECT statement using a relational operator:

```
SELECT * FROM customer WHERE zipcode >= "98000"
```

Avoiding Noninitial Substrings

A filter based on a noninitial substring of a column also requires every value in the column to be tested. Here is an example:

```
SELECT * FROM customer
    WHERE zipcode[4,5] > "50"
```

The optimizer does not use an index to evaluate such a filter even when one exists.

The optimizer uses an index to process a filter that tests an initial substring of an indexed column. However, the presence of the substring test can interfere with the use of a composite index to test both the substring column and another column.

A test of an initial substring can often be rewritten as a relational or BETWEEN test on the whole column, and may be faster when that is done.

Using a Temporary Table to Speed Queries

Building a temporary, ordered subset of a table can sometimes speed up a query. It can help to avoid multiple-sort operations and can simplify the work of the optimizer in other ways.

Using a Temporary Table to Avoid Multiple Sorts

For example, suppose your application produces a series of reports on customers who have outstanding balances, one report for each major postal area, ordered by customer name. In other words, there is a series of queries each of this form (using hypothetical table and column names):

```
SELECT cust.name, rcvbles.balance, ...other columns...
    FROM cust, rcvbles
    WHERE cust.customer_id = rcvbles.customer_id
        AND rcvbls.balance > 0
        AND cust.postcode LIKE "98_ _ _"
    ORDER BY cust.name
```

This query reads the entire **cust** table. For every row with the right postcode, the database server searches the index on **rcvbles.customer_id** and performs a nonsequential disk access for every match. The rows are written to a temporary file and sorted.

This procedure is acceptable if the query is done only once, but this example includes a series of queries, each incurring the same amount of work.

An alternative is to select all customers with outstanding balances into a temporary table, ordered by customer name, like this:

```
SELECT cust.name, rcvbles.balance, ...other columns...
    FROM cust, rcvbles
    WHERE cust.customer_id = rcvbles.customer_id
        AND cvbls.balance > 0
    ORDER BY cust.name
    INTO TEMP cust_with_balance
```

Now you can direct queries against the temporary table in this form:

```
SELECT *
    FROM cust_with_balance
    WHERE postcode LIKE "98_ _ _"
```

Each query reads the temporary table sequentially, but the table has fewer rows than the primary table. No nonsequential disk accesses are performed. No sort is required since the physical order of the table is the desired order. The total effort should be considerably less than before.

There is one possible disadvantage: any changes made to the primary table after the temporary table has been created are not reflected in the output. This is not a problem for most applications, but it might be for some.

Substituting Sorting for Nonsequential Access

Nonsequential disk access is the slowest kind. The SQL language hides this fact, and makes it easy to write queries that access huge numbers of pages nonsequentially. Sometimes you can improve a query by substituting the sorting ability of the database server for nonsequential access.The following example demonstrates this, and also demonstrates how to make numerical estimates of the cost of a query plan.

Imagine a large manufacturing firm whose database includes three tables shown in schematic form in Figure 4-6. (Table diagrams in this form are described in Chapter 8 of this manual. Not all the columns are shown.)

The first table, **part**, contains parts used in the products of the company.

The second, **vendor**, contains data about the vendors who provide the parts. The third, **parven**, records which parts are available from which vendors, and at what price.

part

part_num	part_desc
pk	
100,000 999,999	spiral spanner spotted paint

vendor

vendor_num	vendor_name
pk	
9,100,000 9,999,999	Wrenchers SA Spottiswode

parven

part_num	vendor_num	price
pk, fk	pk, fk	
100,000 999,999	9,100,000 9,999,999	$14.98 $0.066

Figure 4-6   Three tables from a manufacturing database

The following query is run regularly against these tables to produce a report of all available prices:

```
SELECT part_desc, vendor_name, price
    FROM part, vendor, parven
    WHERE part.part_num = parven.part_num
        AND parven.vendor_num = vendor.vendor_num
    ORDER BY part.part_num
```

Although it appears to be a relatively simple three-table join, the query takes too much time. As part of your investigation, you prepare a table showing the approximate sizes of the tables and their indexes, in pages. The table appears in Figure 4-7. It is based on a disk page size of 4,096 bytes. Only approximate numbers are used. The actual number of rows changes often, and only estimated calculations are made in any case.

On further investigation, you learn that the index on **part_num** is clustered, so that the **part** table is in physical order by **part_num**. Similarly, the **vendor** table is in physical order by **vendor_num**. The **parven** table is not in any particular order. The sizes of these tables indicate that the odds of a successful nonsequential access from a buffered page are very poor.

The best query plan for this query (not necessarily the one the optimizer chooses) is to read the **part** table sequentially first, then use the value of **part_num** to access the matching rows of **parven** (about 1.5 of them per part), and then use the value of **parven.vendor_num** to access **vendor** through its index.

Tables	Row Size	Row Count	Rows/Page	Data Pages
part	150	10,000	25	400
vendor	150	1,000	25	40
parven	13	15,000	300	50

Indexes	Key Size	Keys/Page	Leaf Pages
part_num	4	500	20
vendor_num	4	500	2
parven (composite)	8	250	60

Figure 4-7   *Documenting the sizes of tables and indexes in pages*

The resulting number of disk accesses can be estimated as follows:

- 400 pages read sequentially from **part**
- 10,000 nonsequential accesses to the **parven** table, 2 pages each (one index leaf page, one data page), or 20,000 disk pages
- 15,000 nonsequential accesses to the **vendor** table, or 30,000 disk pages

Even a simple join on well-indexed tables can cost 50,400 disk reads. However, this can be improved by breaking the query into three steps using temporary tables as shown in Figure 4-8. (The following solution comes from W.H. Inmon; it is adapted from an example in his book *Optimizing Performance in DB2 Software*, Prentice-Hall 1988.)
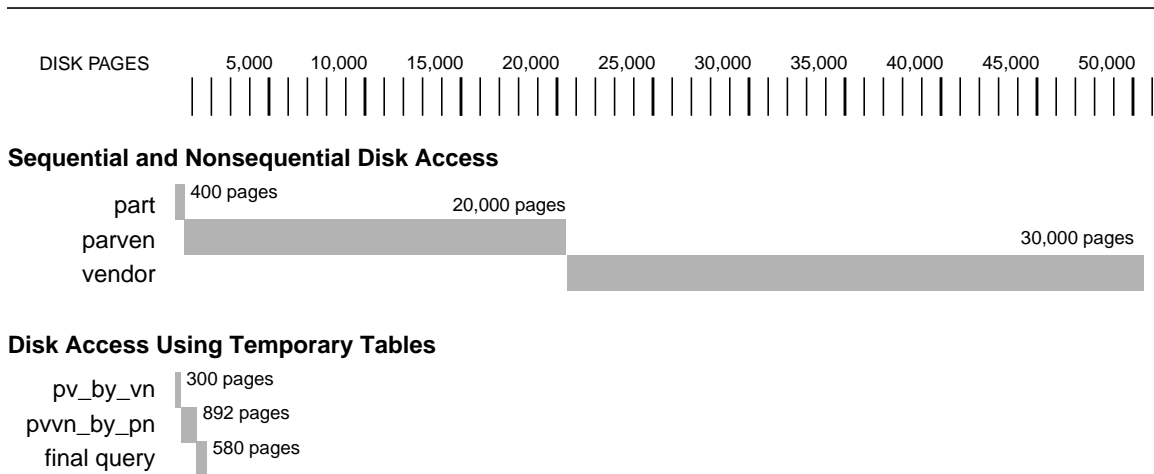
Figure 4-8   ***Breaking a query into three steps using temporary tables***

The first step is to get the data of the **parven** table in **vendor_num** order:

```
SELECT part_num, vendor_num, price
    FROM parven
    ORDER BY vendor_num
    INTO TEMP pv_by_vn
```

This statement reads **parven** sequentially (50 pages), writes a temporary table (50 pages), and sorts it. The cost of the sort, assuming one merge level, is about 200 pages, for a total of 300.

Join this temporary table to **vendor** and put the result in another temporary table, ordered by **part_num**:

```
SELECT pv_by_vn.*, vendor.vendor_name
    FROM pv_by_vn, vendor
    WHERE pv_by_vn.vendor_num = vendor.vendor_num
    ORDER BY pv_by_vn.part_num
    INTO TEMP pvvn_by_pn;
DROP TABLE pv_by_vn
```

This query reads **pv_by_vn** sequentially (50 pages). It accesses **vendor** by way of its index 15,000 times, but as the keys are presented in **vendor_num** sequence, the effect is to read **vendor** sequentially through its index (42 pages).

If the **vendor_name** field is 32 bytes long, the output table has about 95 rows per page and occupies about 160 pages. These pages are written and then sorted, causing 5×160=800 page reads and writes. Thus, this query reads or writes 892 pages in all. Now join its output to **part** for the final result:

```
SELECT pvvn_by_pn.*, part.part_desc
    FROM pvvn_by_pn, part
    WHERE pvvn_by_pn.part_num = part.part_num;
DROP TABLE pvvn_by_pn
```

This query reads **pvvn_by_pn** sequentially (160 pages). It accesses **part** by way of its index 15,000 times, but again the keys are presented in **part_num** sequence so the result is to read **part** sequentially (400 pages) through its index (20 pages). The output is produced in sorted order.

By splitting the query into three steps, and by substituting sorts for indexed access, a query that read 50,400 pages is converted into one that reads and writes approximately 1772 pages, a 30-to-1 ratio.

Note: Informix database servers earlier than version 4.1 do not permit you to use the ORDER BY and INTO TEMP clauses in the same query. With these database servers, you can achieve the preceding solution by selecting INTO TEMP without ordering, then applying the desired ordering through a clustered index. The improvement is, at most, 15 to 1 rather than of 30 to 1.

Summary

Poor performance can come from a number of sources, not solely the SQL operations in a program. Before you focus your attention on the code, take the following steps:

- Examine the application in its context of machines, people, procedures, and organization.
- Understand exactly what the application does, for whom, and why.
- Look for nontechnical solutions in the surrounding context.
- Develop a means of taking repeatable, quantitative measurements of the performance of the application.
- Isolate the time-consuming parts as narrowly as possible.

The performance of an individual query is determined by the optimizer, a part of the database server that formulates the query plan, the order in which tables are read from disk. The optimizer exhaustively lists all possible query plans and estimates the amount of work each causes. It passes the one with the lowest estimate to the database server for execution.

The following operations in a query are time consuming:

- Reading rows from disk sequentially, nonsequentially by ROWID, and nonsequentially through an index
- Reading rows over a network
- Sorting
- Performing some difficult matches of regular expressions

The general method to speed a query is first, to study the query plan of the optimizer as documented by the SET EXPLAIN ON feature, and second, to change the query so that the database server

- Reads fewer pages of data from disk
- Reads pages sequentially rather than nonsequentially
- Avoids sorting or sorts fewer rows or sorts on narrower columns

You can achieve remarkable time savings in this way. Even if no time is saved, the necessary study of the application and the query at least make it clear where the time is being spent.

Statements That Modify Data

5

Chapter Overview

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 a moment 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 itself is in doubt. Obviously, this can have 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 are unable to answer yes to each of these questions, do not panic. Solutions to all these problems are built into the database servers. After an introduction to the statements that modify data, this chapter discusses these solutions. Chapters 8 through 11 of this manual talk about these topics in greater detail.

Statements That Modify Data

The following three statements modify data:

- DELETE

- INSERT

- UPDATE

Although these SQL statements are relatively simple when compared with the more advanced SELECT statements, use them carefully since they do change the contents of the database.

Deleting Rows

The DELETE statement removes any row or combination of rows from a table. There is no way you can recover a deleted row once the transaction is committed. (Transactions are discussed under "Interrupted Modifications" on page 5-21. For now, think of a transaction and a statement as the same thing.)

When deleting a row, you must be careful also to delete any rows of other tables whose values depend on the deleted row. If, however, your database enforces referential constraints, you are not allowed to delete rows when other rows depend on their value. For more information on referential constraints, refer to the section "Referential Integrity" on page 5-19.

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
```

Since this DELETE statement contains no WHERE clause, all rows from the **customer** table are deleted. If you attempt an unconditional delete using the **DB-Access** or **INFORMIX-SQL** menu options, you are warned and asked for confirmation. However, an unconditional delete from within a program is performed 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. For example, you can delete a customer with a specific customer number:

```
DELETE FROM customer WHERE customer_num = 175
```

In this example, since the **customer_num** column has a unique constraint, you are sure that, at most, one row is deleted.

Deleting an Unknown Number of Rows

You can also choose rows based on nonindexed columns, for example:

```
DELETE FROM customer WHERE company = "Druid Cyclery"
```

Since the column tested does not have a unique constraint, this statement might delete more than one row. (Druid Cyclery may have two stores, both with the same name but different customer numbers.)

You can find out how many rows might be affected by a DELETE statement by selecting 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 be sure they are the ones you mean 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

- Insert a new row for another customer named Druid Cyclery
- Delete one or more of the Druid Cyclery rows before you do so
- 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 5-25, and in greater detail in Chapter 7 of this manual.

Another problem you may encounter is a hardware or software failure before the statement finishes. In this case, the database may have deleted no rows, some rows, or all specified rows. The *state* of the database is unknown, which is undesirable. You can prevent this situation by using transaction logging, as discussed in "Interrupted Modifications" on page 5-21.

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 connected by AND and OR, and it may contain subqueries.

Suppose you discover that some rows of the **stock** table were entered with incorrect manufacturer codes. Rather than update them, you want to delete them so they can be reentered. You know that these rows, unlike the correct ones, have no matching rows in the **manufact** table. This allows you to write the following DELETE statement:

```
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. Here is an example of adding 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 identifying the type of merchandise
- **manu_code**, a foreign key to the **manufact** table
- **description**
- **unit_price**
- **unit** (of measure)
- **unit_descr** (characterizing the unit of measure)

Notice that the values listed in the VALUES clause in the preceding example have a one-to-one correspondence with the columns of this 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. The values you supply can include

- Literal numbers
- Literal datetime values
- Literal interval values
- Quoted strings of characters
- The word NULL for a null value
- The word TODAY for today's 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 may not permit duplicate values. If you specify a value that is a duplicate of

one already in such a column, the statement is rejected. Some columns may even *restrict* the possible column values allowed. These restrictions are placed on columns using data integrity constraints. For more information on data restrictions, see the section "Database Privileges" on page 5-15.

Only one column in a table can have the SERIAL data type. The database server will generate values for a serial column. To make this happen, specify the value zero for the serial column and 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 may 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 representing today's date is used. (The format used is specified by the DBDATE environment variable.)

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 you named. The following example inserts a new row into the **stock** table:

```
INSERT INTO stock (stock_num, description, unit_price, manu_code)
    VALUES (115, "tyre pump", 114, "SHM")
```

Notice that only the data for the stock number, description, unit price, and manufacturer code is provided. The database server supplies the 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 does not specify a default value or 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. However, you can list the columns in any order—as long as the values for those columns are listed in the same order.

After the INSERT statement 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 the **unit** and **unit_descr** are blank, indicating that null values are in those two columns. Since the **unit** column permits nulls, one can only guess the number of tire pumps that were purchased for $114. Of course, if a default value of "box" had been specified for this column, then "box" would have been the unit of measure. In any case, when inserting values into specific columns of a table, pay special attention to what data is needed to make sense of 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 (a row is inserted for each row returned by the SELECT statement)
- Calculated values (the VALUES clause only permits constants) since 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 following INSERT statement finds those orders and then inserts a row in **cust_calls** for each order:

```
INSERT INTO cust_calls (customer_num, call_descript)
    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

There are four restrictions on the SELECT statement:

- 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.

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 6 of this manual). You can work around the INTO TEMP clause restriction by first selecting the data you want to insert into a temporary table and then inserting the data from the temporary table using 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, then insert from that temporary table. Or you can apply a physical order to the table using a clustered index after all insertions are done.

The fourth 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. (This saves the database server from getting into an endless loop in which each inserted row is reselected and reinserted.) In some cases, however, you may want to do this. 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 this. 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.

There is a way around this restriction. Select the data you want to insert into a temporary table. Then, select from that table in the INSERT statement. Three statements are required to accomplish this:

```
SELECT stock_num, "NKL" temp_manu, description, unit_price/2
       half_price, unit, unit_descr FROM stock
    WHERE manu_code = "ANZ"
    INTO TEMP anzrows

INSERT INTO stock SELECT * FROM anzrows

DROP TABLE anzrows
```

This SELECT statement takes existing rows from **stock**, 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, there is a risk that one of the rows contains invalid data that will cause the database server to report an error. When this happens, the statement terminates early. Even if no error occurs, there is also a very small risk that there will be a hardware or software failure 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. Since the database is in an unknown state, you cannot tell what to do. The answer lies in using transactions, as discussed under "Interrupted Modifications" on page 5-21.

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 statements 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, refer to the section on "Database Privileges" on page 5-15.

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 it, all rows are modified. The WHERE clause can be quite complicated, so as to select the precise set of rows that need changing. The only restriction on it is that the table that you are updating 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. Here is an example:

```
UPDATE customer
    SET fname = "Barnaby", lname = "Dorfler"
    WHERE customer_num = 103
```

The WHERE clause selects the row to be updated. In the **stores5** database the **customer.customer_num** column is the primary key for that table, so this statement can update one row, at most.

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.

```
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 of them). 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 is updated. 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%, and you must update the **stock** table to reflect this.

```
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). Suppose that 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.

```
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. This is a correlated subquery; because a value from **items** appears in its WHERE clause, it must be executed for every row that is updated.

The second SELECT statement produces a list of the order numbers of unshipped orders. It is an uncorrelated subquery and is executed once.

Impossible Updates

There are restrictions 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 in which the **unit_price** column was incremented by 5%. You *can* 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.

The need to update and query a table at the same time does not arise often in a well-designed database (database design is covered in Chapters 8 through 11 of this manual). However, you may want to do this when a database is first being developed, before its design has been carefully thought through. A typ-

ical 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, which is not allowed in an UPDATE statement or DELETE statement. Chapter 7 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 in this example:

```
UPDATE customer
    SET (fname, lname) = ("Barnaby", "Dorfler")
    WHERE customer_num = 103
```

There is no advantage to writing the statement this way. In fact, it is harder to read since 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. (This idea of collecting routine changes in a separate table and applying them in a batch, outside of prime hours, is one of the performance techniques covered in Chapter 10 of this manual.)

```
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)
```

Notice that 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, it also takes much longer to execute.

Note: In **INFORMIX**-**4GL** *and the embedded SQL programs, you can use record or host variables to update values. For more information on this, refer to Chapter 6 of this manual.*

Database Privileges

There are two levels of privileges in a database: database-level privileges and table-level privileges. When you create a database, you are the only one who can access it until you, as the owner (or DBA) of the database, grant database-level privileges to others. 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 three database-level privileges follow:

- Connect privilege, which allows you to open a database, issue queries, and create and place indexes on temporary tables

- Resource privilege, which allows you to create permanent tables

- DBA privilege, which allows you to perform a number of additional functions as the database administrator

There are seven table-level privileges. However, only the first four are covered here:

- Select privilege, which is granted on a table-by-table basis and allows you to select rows from a table (This privilege can be limited by specific columns in a table.)

- Delete privilege, which allows you to delete rows

- Insert privilege, which allows you to insert rows

- Update privilege, which 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.

The other table-level privileges are needed to modify data. The owners of tables often withhold these privileges or grant them only to specific users. As a result, you may not be able to modify some tables that you can query freely.

Since 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. The Update privileges can be restricted even further to specific columns *in* a table.

Chapter 11 of this manual contains a complete discussion of the granting of privileges from the standpoint of the database designer. A complete list of privileges and a summary of the GRANT and REVOKE statements can be found in Chapter 7 of *The Informix Guide to SQL: Reference.*

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. So, 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 display:

grantor	grantee	tabid	tabauth
tfecit	mutator	101	su-i-x-
tfecit	procrustes	101	s--idx-
tfecit	public	101	s--i-x-

The grantor is the user who *granted* 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 has been 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, execute a DELETE statement), it performs a query similar to the preceding one. If you are not the owner of the table, and if it 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. Or 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 three 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 could harm the data integrity.

Entity Integrity

An entity is any person, place, or thing to be recorded in a database. Each entity represents a table, 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 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 o**rder_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.

Refer to Chapter 8 of this manual for more information on primary keys and entity integrity.

Semantic Integrity

Semantic integrity ensures that data entered into a row reflects an allowable value for that row. This means that the value must be within the *domain*, or allowable set of values, for that column. For example, the **quantity** column of the **items** table only permits numbers. If a value outside the domain can be entered into a column, the semantic integrity of the data is violated.

Semantic integrity is enforced using the following constraints:

- 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 may check for quantities greater than or equal to 1.

For more information on using semantic integrity constraints in database design, refer to "Defining the Domains" on page 9-3.

Referential Integrity

Referential integrity refers to the relationship *between* tables. Since each table in a database must have a primary key, it is possible that this primary key appears 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, in the **stores5** database, 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.

**customer Table
(detail)**

customer_num	fname	lname
103	Philip	Currie
106	George	Watson

**orders Table
(detail)**

order_num	order_date	customer_num
1002	05/21/1991	101
1003	05/22/1991	104
1004	05/22/1991	106

**cust_calls Table
(detail)**

customer_num	call_dtime	user_id
106	1991-06-12 8:20	maryj
110	1991-07-07 10:24	richc
119	1991-07-01 15:00	richc
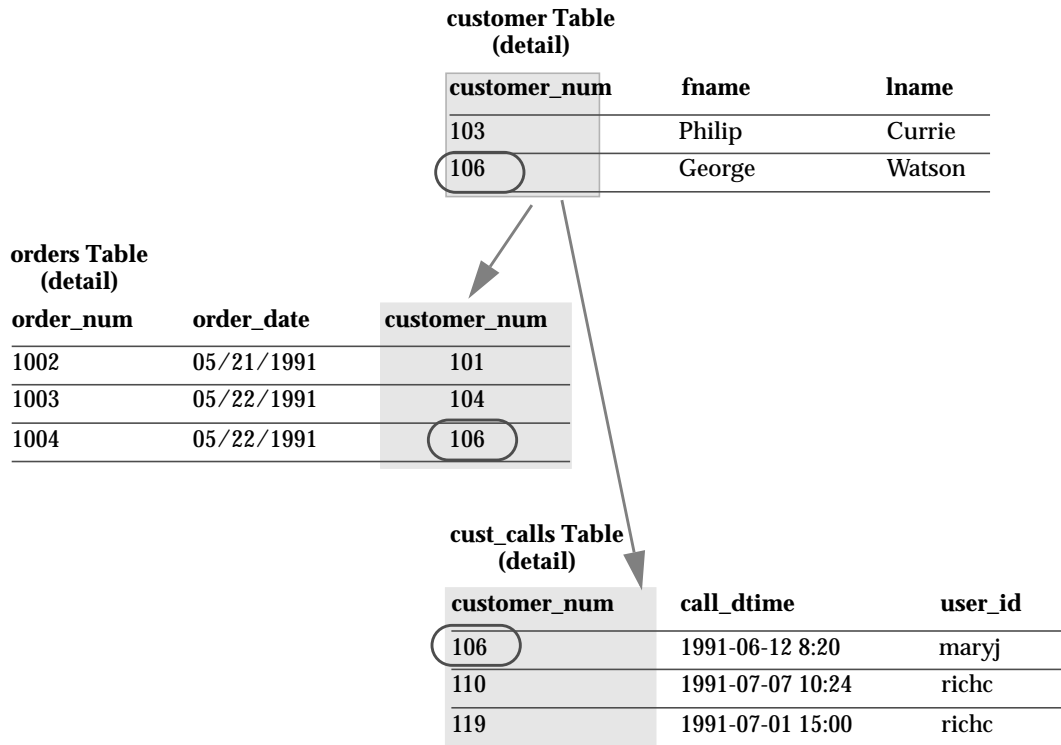
When you delete a row containing a primary key, or update it with a different primary key, you destroy the meaning of any rows that contain that value as a foreign key. Referential integrity is the logical dependency of a foreign key on a primary key. The *integrity* of a row that contains a foreign key depends on the integrity of the row that it *references*—the row that contains the matching primary key.

You can identify primary and foreign keys, and the relationship between them, using the CREATE TABLE and ALTER TABLE statements. For more information on these statements, refer to Chapter 7 of *The Informix Guide to SQL: Reference.* For information on building data models using foreign and primary keys, refer to Chapter 8 of this manual.

Interrupted Modifications

Even if all the software is error-free and all the hardware is utterly reliable, the world outside the computer can still interfere. It is possible for lightning to strike the building, interrupting the electrical supply and stopping the computer in the middle of your UPDATE statement. It is more likely, however, that a disk fills up or a human operator 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 ones are worse. It does not help that they are usually embedded in programs so that you do not see the individual SQL statements being executed. For example, the job of entering a new order in the **stores5** database requires these steps:

- Insert a row in the **orders** table. (This generates an order number.)
- For each item ordered, insert a row in the **items** table.

There are two ways to program an order-entry application. One way is to make it completely interactive so that the program inserts the first row immediately, and then inserts each item as the operator enters data. This is the wrong approach because it exposes the operation to the possibility of many more unforeseen events: the customer's telephone disconnecting, the operator hitting the wrong key, the operator's terminal losing power, and so on.

The right way to build an order-entry application is as follows:

- Accept all the data interactively.
- Validate the data and expand it (by looking 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 all the insertions as fast as possible.

Even when this is done, something 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: *data integrity* is compromised.

The Transaction

The solution to all these potential problems is called the *transaction*. A transaction is a sequence of modifications that either must be accomplished 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 state it was in 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. (This is discussed further in Chapter 7 of this manual.)

The Transaction Log

The database server can keep a journal 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 data in the journal to reverse the changes.

This journal is called a *transaction log*. The log is typically a disk file separate from the database. The action of keeping the transaction up-to-date is called *transaction logging.*

When a transaction fails, the database server uses the contents of the log to undo whatever modifications were made when the transaction started. Many things can make a transaction fail. The program that issues the SQL statements can crash or be terminated, or there might be a hardware or software failure in any other component of the system. As soon as the database server discovers that the transaction failed—which might only be after the computer and the database server are restarted—it returns the database to the state it was in before the transaction began.

Databases do not keep transaction logs automatically. The database administrator must decide whether to make a database use transaction logging. If this is not done, transactions are not available.

Specifying Transactions

The boundaries of transactions are specified with SQL statements. There are two styles for doing this. In the most common style, you specify the start of a multistatement transaction by executing the BEGIN WORK statement. In databases that are created with the MODE ANSI option, there is no need to mark the beginning of a transaction. One is always in effect; you only indicate the end of each transaction.

In both styles, you specify the end of a successful transaction by executing the COMMIT WORK statement. This statement tells the database server that you reached the end of a series of statements that must all succeed together. The database server does whatever is necessary to make sure that all modifications are properly completed and committed to disk.

It is also possible for a program to cancel a transaction deliberately. It does so by executing the ROLLBACK WORK statement. This statement asks the database server to cancel the current transaction and undo any changes.

For example, an order-entry application can use a transaction when creating a new order in the following way:

- 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 returned by the database server.
- If there were no errors, execute COMMIT WORK, otherwise execute ROLLBACK WORK.

If any external failure prevents the transaction from being completed, the partial transaction is rolled back when the system is restarted. In all cases, the database is in a predictable state: either the new order is completely entered, or it is not entered at all.

Archives and Logs

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 *archive* copies.

The transaction log complements the archive copy of a database. Its contents are a history of all modifications that occurred since the last time the database was archived. If it should ever be necessary to restore the database from the archive copy, the transaction log can be used to roll the database forward to its most recent state.

Archiving Simple Databases (INFORMIX-SE)

If a database is stored in operating system files (**INFORMIX-SE**), archive copies are made using the normal methods for making backup copies in your operating system. There are only two special considerations for databases.

The first is a practical consideration: a database can grow to great size. It may become the largest file or set of files in the system. It also can be awkward or very time consuming to make a copy of it. You may need a special procedure for copying the database, separate from the usual backup procedures, and the job may not be done as frequently.

The second consideration is the special relationship between the database and the transaction log file. An archive copy is an image of the database at one instant. The log file contains the history of modifications that were made since one instant. It is important that those two instants are identical; in other words, it is important to start a new transaction log file immediately upon making an archive copy of the database. Then, if you must restore the database from the archive tape, the transaction log contains exactly the history needed to bring it forward in time from that instant to the latest update.

The statement that applies a log to a restored database is ROLLFORWARD DATABASE. You start a new log file by using whatever operating system commands are needed to delete the file and re-create it empty, or simply to set the length of the file to zero.

A transaction log file can grow to extreme size. If you update a row ten times, there is still just one row in the database—but there are ten update events recorded in the log file. If the size of the log file is a problem, you can start a fresh log. Choose a time when the database is not being updated (so no transactions are active), and copy the existing log to another medium. That copy represents all modifications for some period of time; preserve it carefully. Then start a new log file. If you ever have to restore the database, you must apply all the log files in their correct sequence.

Archiving INFORMIX-OnLine

The **INFORMIX-OnLine** database server contains elaborate features to support archiving and logging. They are described in the *INFORMIX-OnLine Administrator's Guide.*

Conceptually, the facilities of **INFORMIX**-**OnLine** are similar to those already described, but they are more elaborate for the following reasons:

- **INFORMIX**-**OnLine** has very stringent requirements for performance and reliability (for example, it supports making archive copies while databases are in use).

- It manages its own disk space.

- It performs logging concurrently for all databases using a limited set of log devices.

These facilities are usually managed from a central location, so the users of the databases never have to be concerned with them.

If you want to make a personal archive copy of a single database or table that is held by **INFORMIX**-**OnLine**, you can do it with the **tbunload** utility. This program copies a table or a database to tape. Its output consists of binary images of the disk pages as **INFORMIX**-**OnLine** held them. As a result, the copy can be made very quickly, and the corresponding **tbload** program can restore the file very quickly. However, the data format is not meaningful to any other programs.

Concurrency and Locks

If your database is contained in a single-user workstation, with no network connecting it to other machines, concurrency does not concern you. But 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. This is *concurrency*: 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 there are controls 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 seemed they were entered successfully.

The database server prevents errors of this kind by imposing 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 two commands to control the effect that locks have on your data access: SET LOCK MODE and SET ISOLATION. The details of these statements are best understood after a discussion on the use of *cursors* from within programs; they are covered in Chapters 6 and 7 of this manual.

Summary

Database access is regulated by the privileges that the database owner grants 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.

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 containing specified column values, or you can insert a block of rows generated by a SELECT statement.

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 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; you use the other form when the new values are generated as a set from a SELECT statement or a record variable.

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 following an error. The transaction log also extends the periodically made archive copy of the database, so that if the database must be restored, it can be brought right up to its most recent state.

SQL in Programs

6

Chapter Overview

In the examples in the chapters thus far, 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 how things are. 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 some program requests it.

DB-Access and **INFORMIX-SQL** are two programs that perform this requesting, and format and display the data. They are designed to give you interactive access to SQL. Other programs are written for a specific application. Such programs can be written in any of several languages.

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, since the details of the process are slightly different in every language, you must become familiar with the manual for the Informix Embedded SQL (**ESQL**) product specific to that language.

SQL in Programs

You can write a program in any of several languages and mix SQL statements in among the other statements of the program, just as if they were ordinary statements of that programming language. The SQL statements are said to be *embedded* in the program, and the program is said to contain *embedded SQL*, often abbreviated **ESQL**.

Informix produces **ESQL** products for these programming languages:

- C
- COBOL
- FORTRAN
- Ada (version 4.0 only)

All **ESQL** products work in a similar way, as shown in Figure 6-1. You write a source program in which you treat SQL statements as executable code. Your source program is processed by an **ESQL** *preprocessor*, a program that locates the embedded SQL statements and converts them into a series of procedure calls and special data structures.

ESQL source program   ESQL preprocessor   Source program with procedure calls   Language compiler   Executable program
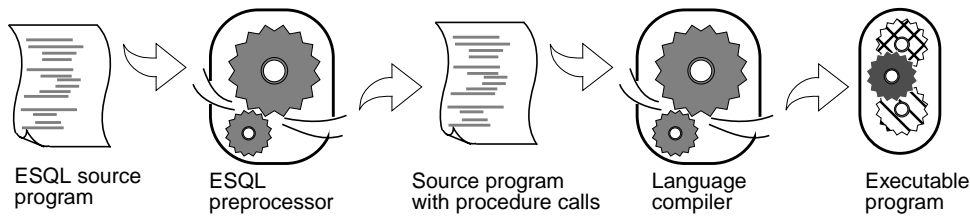
Figure 6-1   Overview of processing a program with embedded SQL statements

The converted source program then passes through the programming language compiler. The compiler output becomes an executable program after it is linked with a library of **ESQL** procedures. When the program runs, the **ESQL** library procedures are called; they set up communications with the database server to carry out the SQL operations.

Whereas **ESQL** products allow you to embed SQL in the host language, some languages have SQL as a natural part of their statement set. **INFORMIX-4GL** incorporates the SQL language as a natural part of the fourth-generation language it supports. Informix Stored Procedure Language (SPL) also uses SQL as a natural part of its statement set. You use **INFORMIX-4GL** or an **ESQL** 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

There are two ways in which you can introduce SQL statements into a program. The simpler and more common way is by *static embedding,* which means that the SQL statements are written as part of the source program text. The statements are *static* because they are a fixed part of the source text.

Dynamic Statements

Some applications require the ability to compose SQL statements 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.

This can be done with *dynamic* SQL, in which 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 program source text; they are constructed in memory during execution.

Program Variables and Host Variables

Application programs can use program variables within SQL statements. In **INFORMIX-4GL** and 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. Figure 6-2 shows a program variable in **INFORMIX-4GL**.

```
MAIN
.
.
.
DEFINE drop_number INT
LET drop_number = 108
DELETE FROM items WHERE order_num = drop_number
.
.
.
```

Figure 6-2   Using a program variable in INFORMIX-4GL

Figure 6-3 shows a program variable in SPL.

```
CREATE PROCEDURE delete_item (drop_number INT)
.
.
.
DELETE FROM items WHERE order_num = drop_number
.
.
.
```

Figure 6-3   *Using a program variable in SPL*

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 being a "guest" in the program.

Here is a DELETE statement as it might appear when embedded in a COBOL source program:

```
EXEC SQL
    DELETE FROM items
        WHERE order_num = :o-num
END-EXEC.
```

The first and last lines mark off embedded SQL from the normal COBOL statements. Between them you see an ordinary DELETE statement, just as it was described in Chapter 5. When this part of the COBOL program is executed a row, or rows, of the **items** table is deleted.

The statement contains one new feature. It compares the **order_num** column to an item written as **:o-num**. This is the name of a host variable.

Each **ESQL** product has a means of delimiting the names of host variables when they appear in the context of an SQL statement. In COBOL, host variable names are designated with an initial colon. The example statement asks the database server to delete rows in which the order number equals the current contents of the host variable named **:o-num**. This is presumably a numeric variable that is declared and assigned a value earlier in the program.

The same statement embedded in a FORTRAN program looks like this:

```
EXEC SQL
    DELETE FROM items
        WHERE order_num = :onum
```

The same statement embedded in an Ada program looks like this:

```
EXEC SQL
    DELETE FROM items
        WHERE order_num = $onum;
```

In **INFORMIX-ESQL/Ada**, a host variable is indicated by a leading currency symbol, and statements are ended with a semicolon. The same is true of **INFORMIX-ESQL/C**, where the corresponding DELETE statement is written as follows:

```
$   delete from items
        where order_num = $onum;
```

In **INFORMIX-ESQL/C**, an SQL statement can be introduced with either a leading currency symbol or the words EXEC SQL.

But these differences of syntax are trivial; the essential points in all languages (an embedded language, **INFORMIX-4GL**, or SPL) are as follows:

- You can embed SQL statements in a source program as if they were executable statements of the host language.
- You can 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 under a multitude of interfaces, and you can embellish its functions in a multitude of 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.

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. Also, as a later section shows, host variables receive data that is returned by the database server.

The SQL Communications Area

The database server always returns a result code, and possibly other information about the effect of the 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 6-4 and Figure 6-5. The syntax you use to describe a data structure like the SQLCA, as well as the syntax you use to refer to a field in it, depends on the programming language you are using.

integer

SQLCODE

0	Success.
100	No more data/not found.
negative	Error code.

array of 6 integers

SQLERRD

first	Following successful prepare of a SELECT, UPDATE, INSERT, or DELETE statement, or after a select cursor is opened, this field contains the estimated number of rows affected. It is not used in **INFORMIX-ESQL/Ada**.
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.
	Following a successful insert operation of a single row, this field contains the value of a generated serial number for that row.
third	Following a successful, multirow insert, update, or delete operation, this field contains the count of rows processed.
	Following a multirow insert, update, or delete operation that ends with an error, this field contains the count of rows successfully processed before the error was detected.
fourth	Following successful prepare of 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. It is not used in **INFORMIX-ESQL/Ada**.
fifth	Following an error in a PREPARE statement, this field contains the offset in the statement text where the error was detected. (The PREPARE statement is discussed under "Preparing a Statement" on page 6-28.)
sixth	Following 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 processed.

character (8)

SQLERRP

Internal use only.

Figure 6-4   The uses of SQLCODE and SQLERRD

array of 8
characters

SQLAWARN

	When Opening a Database:
first	Set to *W* when any field is set to *W*. If this field is blank, the others need not be checked.
second	Set to *W* when the database now open uses a transaction log.
third	Set to *W* when the database now open is ANSI-compliant.
fourth	Set to *W* when the database server is **INFORMIX-OnLine**.
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).
sixth	Not used.
seventh	Not used.
eighth	Not used.

	All Other Operations:
first	Set to *W* when any other field is set to *W*.
second	Set to *W* when a column value is truncated when it is fetched into a host variable.
third	Set to *W* when an aggregate function encounters a null value.
fourth	On a select or on opening a cursor, 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.
fifth	After preparing an UPDATE or DELETE statement, set to *W* when the prepared statement has no WHERE clause, and so affects an entire table.
sixth	Set to *W* following execution of a statement that does not use ANSI-standard SQL syntax (provided that the DBANSIWARN environment variable is set).
seventh	Not used.
eighth	Not used.

character
(71)

SQLERRM

Contains the error message.

Figure 6-5   The uses of SQLAWARN

In particular, the subscript by which you name one element of the SQLERRD and SQLAWARN arrays differs: array elements are numbered starting with zero in **INFORMIX-ESQL/C**, but starting with one in the other languages. In this discussion, the fields are named using unambiguous words like *third*, and you must translate into the syntax of your programming language.

The 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 shown in Figure 6-4. 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.

In **INFORMIX-4GL**, SQLCODE is also accessible under the name STATUS.

End of Data

The database server sets SQLCODE to 100 when the statement is performed correctly but no rows were found. For example, after the program fetches all the rows that a SELECT statement can produce, the database server sets SQLCODE to 100 to say *end of data*. In an ANSI-compliant database, if the WHERE clause of a DELETE, INSERT, or UPDATE statement matches no rows, 100 is set to show that no rows are found.

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.

Some error codes that can be reported in SQLCODE reflect general problems, and the database server can set a more detailed code in the second field of SQLERRD. This second code reveals the low-level error encountered by the database server I/O routines or by the underlying operating system.

The SQLERRD Array

The integers in the array named SQLERRD are set to different values following different statements. The first and fourth elements of the array are only used in **INFORMIX-4GL**, **INFORMIX-ESQL/C**, **INFORMIX-ESQL/COBOL**, and **INFORMIX-ESQL/FORTRAN**. The fields are used as shown in Figure 6-4.

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 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 **INFORMIX-SQL** use this feature to position the cursor when you ask to modify a statement after an error.)

The SQLAWARN Array

The eight character fields in the SQLAWARN array are set to either a blank or a W to indicate a variety of special conditions. Only the first six are used. Their meanings depend on the statement just executed.

There is one set of warning flags that appears when a database is just opened, that is, following a 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 during the statement, events that might not be reflected by SQLCODE.

Both sets of SQLAWARN values are summarized in Figure 6-5.

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 later in this chapter.

To retrieve a single row of data, simply embed a SELECT statement in your program. Here is an example as it can be written using **INFORMIX-ESQL/C**:

```
$   select avg (total_price)
        into $avg_price
        from items
        where order_num in
            (select order_num from orders
                where order_date < date("6/1/91"));
```

The INTO clause is the only detail that distinguishes this statement from any example in Chapter 2 or 3. 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. In this **INFORMIX-4GL** example, host variables are used in two ways, as receivers of data and in the WHERE clause:

```
DEFINE cfname, clname, ccompany CHAR(20)
DEFINE cnumbr INTEGER
LET cnumbr = 104
SELECT fname, lname, company
    INTO cfname, clname, ccompany
    FROM customer
    WHERE customer_num = cnumbr
```

Since the **customer_num** column has a unique index (implemented through a constraint), this query returns only one row. If a query produces more than one row of data, the database server cannot return any data at all. 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 SQLAWARN.

Data Type Conversion

This example retrieves the average of a DECIMAL column, which is itself a DECIMAL value. However, the host variable into which it is placed is *not* required to have that data type.

```
$    select avg (total_price) into $avg_price
         from items;
```

The declaration of the receiving variable **avg_price** in this example of **ESQL/C** code is not shown. It 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 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 used by the receiving variables. Almost any conversion is allowed, although some conversions cause a loss of precision. The results of the preceding example differ depending on the data type of the receiving host variable, as described in the following list:

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 and the second field of SQLAWARN is set to W.

Dealing 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 of recognizing a null item to avoid processing it as data.

Indicator variables meet this need in the embedded-language products. 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.

```
$    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);
```

In this **INFORMIX**-**ESQL/C** example, a single row is selected and a single value is retrieved into the host variable **op_date**. Since 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 that is the case, this program uses a C function to assign a default value to the host variable. (The function **rstrdate** is part of the **INFORMIX**-**ESQL/C** product.)

The syntax you use to associate an indicator variable differs with the language you are using, but the principle is the same in all. However, indicator variables are not used explicitly in **INFORMIX**-**4GL** or in SPL, since in those languages null values are supported for variables. In **4GL**, the preceding example is written as follows:

```
SELECT paid_date
    INTO op_date
    FROM orders
    WHERE order_num = the_order
IF op_date IS NULL THEN
    LET op_date = date ("01/01/1900")
END IF
```

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 is signalled by a code of 100 in SQLCODE (or **sqlca.sqlcode** as it is also known in **ESQL/C**) following a SELECT statement. This code indicates an error or a normal event, depending entirely on your application. If you are sure that there ought to be a row or rows—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 are selecting a row based on a key supplied by a user or some other source that is less reliable than a program, a lack of data can be a normal event.

End of Data with Aggregate Functions

If your database is not ANSI-compliant, the end-of-data return code, 100, is set in SQLCODE only following SELECT statements. (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 7 of this manual.)

A SELECT statement that selects an aggregate function such as SUM, MIN, or AVG always succeeds in returning at least one row of data. This is true 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 based on no rows and one 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. Then you can work out the following three 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

Serious Errors

The database server rarely reports a serious problem. Negative return codes occur infrequently and at unexpected times, but they cannot be neglected on that account.

For example, a query can return error -206, meaning `table name is not in the database`. This happens if someone dropped the table since the program was written or if, through some error of logic or mistake in input, the program opened the wrong database. Such errors can occur, and the program must deal with them.

Using Default Values

There are many ways to handle these inevitable errors. In some applications, there are more lines of code to handle errors than for normal situations. In the examples in this section, however, one of the simplest methods, the default value, should work.

```
avg_price = 0; /* set default for errors */
$   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 */
```

This example deals with the following considerations:

- If the query selects some non-null rows, the correct value is returned and used. This is the expected and most frequent result.
- 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.

Here is an expansion of an earlier **INFORMIX-4GL** example that displays default values if it cannot find the company the user requests:

```
DEFINE cfname, clname, ccompany CHAR(20)
DEFINE cnumbr INTEGER
PROMPT "Enter the customer number: " FOR cnumbr
LET cfname = "unknown"
LET clname = "person"
LET ccompany = "noplace"
SELECT fname, lname, company
    INTO cfname, clname, ccompany
    WHERE customer_num = cnumbr
DISPLAY cfname, " ", clname, " at ", ccompany
```

This query does not use aggregates, so if there is no row matching the user-specified customer number, SQLCODE is set to 100 and the host variables remain unchanged.

Retrieving Multiple Rows

When there is any chance 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 general sequence of program operations is as follows:

1. The program *declares* the cursor and its associated SELECT statement. This merely allocates storage to hold the cursor.

2. The program *opens* the cursor. This 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.

These operations are performed with SQL statements named DECLARE, OPEN, FETCH, and CLOSE.

Declaring a Cursor

A cursor is declared using the DECLARE statement. This statement gives the cursor a name, specifies its use, and associates it with a statement. Here is a simple example in **INFORMIX-4GL**:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
    INTO o_num, i_num, s_num
    FROM items
```

The declaration gives the cursor a name (**the_item** in this case) and associates it with a SELECT statement. (Chapter 7 of this manual discusses how a cursor also can be associated with an INSERT statement.)

The SELECT statement in this example contains an INTO clause. That is one of two ways in which you can specify the variables that receive data.

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 once through the **items** table. Cursors can be declared to read both backward and forward (see "Cursor Input Modes" on page 6-21). This cursor, since it lacks a FOR UPDATE clause, probably is used only to read data, not to modify it. (The use of cursors to modify data is covered in Chapter 7 of this manual.)

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 SQLCODE. Here is the OPEN statement in **INFORMIX-4GL**:

```
OPEN the_item
```

Since this is the first time that the database server has seen the query, it is the time when many errors are detected. After opening the cursor, the program should test SQLCODE. If it contains a negative number, the cursor is not usable. There may be an error in the SELECT statement, or some other problem may be preventing the database server from executing the statement.

If SQLCODE contains a zero, the SELECT statement is syntactically valid and the cursor is ready for 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 also can name the host variables to receive the data. Here is the **INFORMIX-4GL** example completed:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        INTO o_num, i_num, s_num
        FROM items
OPEN the_item
WHILE sqlcode = 0
    FETCH the_item
    IF sqlcode = 0 THEN
        DISPLAY o_num, i_num, s_num
    END IF
END WHILE
```

Detecting End of Data

In this 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, there is also a test of SQLCODE (under its **4GL** name of **status**) within the loop. 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. Here is another way to write the same loop:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        FROM items
OPEN the_item
IF sqlcode = 0 THEN
    FETCH the_item -- fetch first row
END IF
WHILE sqlcode = 0
    DISPLAY o_num, i_num, s_num
    FETCH the_item
END WHILE
```

In this version the case of zero returned rows is handled early, so there is no second test of **sqlcode** 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. It must appear in either the SELECT or the FETCH statements, but not both. Here is the example reworked to specify host variables in the FETCH statement:

```
DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
        FROM items
OPEN the_item
WHILE status = 0
    FETCH the_item INTO o_num, i_num, s_num
    IF status = 0 THEN
        DISPLAY o_num, i_num, s_num
    END IF
END WHILE
```

The second form has the advantage that different rows can be fetched into different variables. For example, you can 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. Hence, it can only read through a table once each time it is opened. A scroll cursor can fetch the next row or any prior row, so it can read rows multiple times. Here is a sequential cursor declared in **INFORMIX-ESQL/C**:

```
EXEC SQL DECLARE pcurs CURSOR FOR
    SELECT customer_num, lname, city
        FROM customer;
```

After it is opened, this cursor can be used only with a sequential fetch that retrieves the next row of data.

```
EXEC SQL FETCH p_curs INTO $cnum, $clname, $ccity;
```

Each sequential fetch returns a new row.

A scroll cursor is declared with the keyword SCROLL, as in this example from **INFORMIX-ESQL/FORTRAN**:

```
 EXEC SQL DECLARE s_curs SCROLL CURSOR FOR
+    SELECT order_num, order_date FROM orders
+        WHERE customer_num > 104
```

A scroll cursor can be used with a variety of fetch options. The ABSOLUTE option specifies the rank number 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**. It is also possible to fetch the current row again, or to fetch the first row and then scan through the entire list again. However, these features are obtained at a price, as described in the next section.

The 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 is a true picture of the situation 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 can be very easy to do, or it can require a great deal of work and time.

```
DECLARE easy CURSOR FOR
    SELECT fname, lname FROM customer
        WHERE state = "NJ"
```

Since this cursor queries only a single table in a simple way, the database server can very quickly discover whether any rows satisfy the query and find the first one. The first row is the only row it finds at this time. The rest of the rows in the active set remain unknown.

```
DECLARE hard 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 (this is discussed in Chapter 4 of this manual), but in general 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 know 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 opening the cursor. Afterward, it could tell the program exactly how many rows the active set contains. This information is not made available, however. One reason is that you can never be sure which method the optimizer uses. If it can avoid sorts and temporary tables, it does; but very small changes in the query, in the sizes of the tables, or in the available indexes can change its methods.

The Active Set for a Sequential Cursor

The database server wants to tie up as few resources as possible in maintaining 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.

The Active Set for a Scroll Cursor

All the rows in the active set for a scroll cursor must be retained until the cursor is closed. This is because the database server cannot be sure which row the program asks 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.

The Active Set and Concurrency

When only one program is using a database, the members of the active set cannot change. This is the situation in most desktop workstations, 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 at the same time.

When other programs can update the tables while your cursor is open, the idea of the active set becomes less useful. Your program can only see 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, there is the problem of *stale data*. 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 may seem unsettling at first, but as long as your program only reads the data, there is no such thing as stale data, or rather, all data is equally stale. No matter when it is taken, 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, there is no practical difference between changes that occur while the program is running, and changes that are saved and applied the instant 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. The next chapter 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 is the parts-explosion problem, sometimes called Bill of Materials processing. At the heart of this problem is a recursive relationship between objects so that 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. 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 #123400 is an assembly of nine parts, there are nine rows with 123400 in the first column and other part numbers in the second.

CONTAINS

PARENT	CHILD	
FK NN	FK NN	
123400	432100	
432100	765899	

The parts-explosion problem is this: given a part number, produce a list of all parts that are components of that part. A sketch of one solution, as implemented in **INFORMIX-4GL**, is shown in Figure 6-6.

```
DEFINE part_list ARRAY[200] OF INTEGER
FUNCTION boom (top_part)
    DEFINE this_part, child_part INTEGER
    DEFINE next_to_do, next_free SMALLINT
    DECLARE part_scan CURSOR FOR
        SELECT child INTO child_part FROM contains
            WHERE parent = this_part

    LET next_to_do = 1
    LET part_list[next_to_do] = top_part
    LET next_free = 2

    WHILE next_to_do < next_free
        this_part = part_list[next_to_do]
        FOREACH part_scan
            LET part_list[next_free] = child_part
            LET next_free = next_free + 1
        END FOREACH
        LET next_to_do = next_to_do + 1
    END WHILE
    RETURN next_free - 1
END FUNCTION
```

Figure 6-6   *A breadth-first algorithm to generate a parts explosion*

Technically speaking, each row of the **contains** table is the head node of a directed acyclic graph, or tree. The function in Figure 6-6 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. That is very easy to implement using the **INFORMIX-4GL** statement FOREACH, which opens a cursor, iterates once for each row in the selection set, and closes the cursor.

This is the heart of the parts-explosion problem, but it is by no means a complete solution. For example, the program in Figure 6-6 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 quite a bit 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 there can be up to four generations of parts 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.) You can extend this solution to more levels by selecting additional nested outer joins of the **contains** table.You also can revise this solution to return counts of the number of parts at each level.

Dynamic SQL

While static SQL is extremely useful, it does require 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.

This is no problem when you write a program to perform a specific, 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 the ability 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 the contents of the statement can determine user input. This is done in three steps:

1. The program assembles the text of an SQL statement as a character string, 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.

DB-Access, the utility that you use to explore SQL interactively, is an **INFORMIX-ESQL/C** program that constructs, prepares, and executes SQL statements dynamically. For example, it enables its users to specify the columns of a table using simple, interactive menus. When the user is 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.

This leads to two restrictions. First, if it is a SELECT statement, it cannot include the INTO clause. That 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. Here is an example in **INFORMIX-ESQL/C**:

```
$    prepare query_2 from
        "select * from orders
            where customer_num = ? and
                order_date > ?";
```

There are two question marks in this example, indicating 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 ones that cannot be prepared 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 SQLAWARN to see if you used a WHERE clause (see "The SQLA-WARN Array" on page 6-12).

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 may contain multiple SQL statements, separated by semicolons. A fairly complex example in **INFORMIX-ESQL/COBOL** is shown in Figure 6-7.

```
EXEC SQL
    MOVE "  BEGIN WORK;
            UPDATE account
                SET balance = balance + ?
                    WHERE acct_number = ?;
            UPDATE teller
                SET balance = balance + ?
                    WHERE teller_number = ?;
            UPDATE branch
                SET balance = balance + ?
                    WHERE branch_number = ?;
            INSERT INTO history VALUES(timestamp, values);
            COMMIT WORK"
    TO :BIG-QUERY.
END-EXEC.

EXEC SQL
    PREPARE BIG-Q FROM :BIG-QUERY
END-EXEC.
```

Figure 6-7   *Preparing a string containing five SQL statements*

When this list of statements is executed, host variables must provide values for six place-holding question marks. While it is more complicated to set up a multistatement list, the 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 any number of times. Statements other than SELECT statements, and SELECT statements that return only a single row, are executed with the EXECUTE statement.

Figure 6-8 shows how **INFORMIX-ESQL/C** prepares and executes a multistatement update of a bank account:

```
$char bigquery[15] = "begin work";
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);
stcat ("commit work;", bigquery);

$PREPARE bigq FROM $bigquery;

$EXECUTE bigq USING $delta, $acct_number, $delta,
    $teller_number, $delta, $branch_number;
```

Figure 6-8   *Preparing and executing a multistatement operation in* **ESQL/C**

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.

Using Prepared SELECT Statements

A dynamically prepared SELECT statement cannot simply be executed; it might produce more than one row of data, and the database server, not knowing which row to return, produces an error code.

Instead, a dynamic SELECT statement is attached to a cursor. Then, the cursor is opened and used in the usual way. The cursor to be used with a prepared statement is declared for that statement name. Here is an example from **INFORMIX-4GL**:

```
LET select_2 = "select order_num, order_date from orders ",
    "where customer_num = ? and order_date > ?"

PREPARE q_orders FROM select_2

DECLARE cu_orders CURSOR FOR q_orders

OPEN cu_orders USING q_c_number, q_o_date

FETCH cu_orders INTO f_o_num, f_o_date
```

The following list identifies the stages of processing in this example:

1. A character string expressing a SELECT statement is placed in a program variable. It employs two place-holding question marks.

2. The PREPARE statement converts the string into a data structure that can be executed. The data structure is associated with a name, **q_orders**.

3. A cursor named **cu_orders** is declared and associated with the name of the prepared statement.

4. When the cursor is opened, the prepared statement is executed. The USING clause in the OPEN statement provides the names of two host variables whose contents are substituted for the question marks in the original statement.

5. The first row of data is fetched from the open cursor. The INTO clause of the FETCH statement specifies the host variables that are to receive the fetched column values.

Later, the cursor can be closed and reopened. While the cursor is closed, a different SELECT statement can be prepared under the name **q_orders**. In this way, a single cursor can be used to fetch from different SELECT statements.

Dynamic Host Variables

In the embedded-language products, which support dynamically allocated data objects, it is possible to take dynamic statements one step further. It is possible to dynamically allocate the host variables that receive column data.

This 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 or an INSERT, the DESCRIBE statement also returns information about the selected or inserted values to a data structure. The data structure is a predefined data structure allocated for this purpose and known as a system descriptor area. If you are using **INFORMIX-ESQL/C**, the data structure can be an **sqlda** pointer 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.

Using 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 belonging to the program. This space is released when the program terminates, but in some programs you might want to release it earlier.

You can use the FREE statement to release this space. It 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.

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 EXECUTE IMMEDIATE statement takes a character string and in one operation prepares it, executes it, and frees the storage.

```
$    EXECUTE IMMEDIATE "drop index my_temp_index";
```

This makes it easy to write simple SQL operations. However, since no USING clause is allowed, the EXECUTE IMMEDIATE statement cannot be used for SELECT statements.

Embedding Data Definition

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 just once, but queried and updated many times.

The creation of a database and its tables is generally done interactively, using **DB-Access** or **INFORMIX-SQL**. These tools can also be driven from a file of statements, so that the creation of a database can be done with one operating system command. (This is discussed in Chapter 10 of this manual.)

Embedding Grant and Revoke Privileges

One task related to data definition is done repeatedly: the granting and revoking of privileges. The reasons for this are discussed in Chapter 11. Since privileges must be granted and revoked frequently, and since this might be done by persons 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 three 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, or command-line parameters, or a file) but none of them 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 prepare and execute the assembled statement. Program input can be incorporated into the prepared statement as characters.

Figure 6-9 shows a function in **INFORMIX**-**4GL** that assembles a GRANT statement from the function parameters, then prepares and executes it.

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
    DEFINE  priv_to_grant char(100),
            table_name char(20),
            user_id char(20),
            grant_stmt char(200)
    LET grant_stmt = " GRANT ", priv_to_grant,
                    " ON ", table_name,
                    " TO ", user_id
    WHENEVER ERROR CONTINUE
    PREPARE the_grant FROM grant_stmt
    IF status = 0 THEN
        EXECUTE the_grant
    END IF
    IF status <> 0 THEN
     DISPLAY "Sorry, got error #", status, "attempting:"
        DISPLAY "      ",grant_stmt
    END IF
    FREE the_grant
    WHENEVER ERROR STOP
END FUNCTION
```

Figure 6-9   ***A 4GL function that builds, prepares, and executes a GRANT statement***

The opening statement defines the name of the function and the names of its three parameters.

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
```

The DEFINE statement defines the parameters and one additional variable local to the function. All four are character strings of various lengths.

```
DEFINE  priv_to_grant char(100),
        table_name char(20),
        user_id char(20),
        grant_stmt char(200)
```

The variable **grant_stmt** holds the assembled GRANT statement, which is created by concatenating the parameters and some constants.

```
LET grant_stmt ="GRANT ", priv_to_grant,
                " ON ", table_name,
                " TO ", user_id
```

In **INFORMIX-4GL**, the comma is used to concatenate strings. This assignment statement concatenates the following six character strings:

- "GRANT"
- The parameter specifying the privileges to be granted
- "ON"
- The parameter specifying the table name
- "TO"
- The parameter specifying the user.

The result is a complete GRANT statement composed partly of program input. The same feat can be accomplished in other host languages using different syntax.

```
WHENEVER ERROR CONTINUE
PREPARE the_grant FROM grant_stmt
```

If the database server returns an error code in SQLCODE, the default action of an **INFORMIX-4GL** program is to terminate. However, errors are quite likely when you prepare an SQL statement composed of user-supplied parts, and program termination is a poor way to diagnose the error. In the preceding code, the WHENEVER statement prevents termination. Then the PREPARE statement passes the assembled statement text to the database server for parsing.

If the database server approves the form of the statement, it sets a zero return code. This does not guarantee that the statement is executed properly; it only means that the statement has correct syntax. It might refer to a nonexistent table or contain many other kinds of errors that can only be detected during execution.

```
IF status = 0 THEN
    EXECUTE the_grant
END IF
```

If the preparation is successful, the next step is to execute the prepared statement. The remainder of the function in Figure 6-9 displays an error message if anything goes wrong. As written, it makes no distinction between an error from the PREPARE operation and one from the EXECUTE operation, and it does not attempt to interpret the numeric error code, leaving it to the user to interpret.

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.

Programs That Modify Data

7

Chapter Overview

The preceding chapter introduced the idea of putting SQL statements, especially the SELECT statement, into programs written in other languages. This enables a program to retrieve rows of data from a database.

This chapter covers the issues that arise when a program needs to modify the database by deleting, inserting, or updating rows. As in Chapter 6 of this manual, the aim is to prepare you for reading the manual for the Informix **ESQL** or **4GL** product you are using.

The general use of the INSERT, UPDATE, and DELETE statements is covered in Chapter 5 of this manual. This chapter examines their use from within a program. It is quite easy to put the statements in a program, but it can be quite difficult to handle errors and to deal with concurrent modifications from multiple programs.

Using DELETE

A program deletes rows from a table by executing 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 5 of this manual; the problem is the same when deletions are made from within a program.

Direct Deletions

You can embed a DELETE statement in a program. Here is an example using **INFORMIX-ESQL/C**:

```
$ 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 the SQLCA, 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 it 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 negative numbers 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 they prevented the database server from beginning the operation at all. 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. Since the issue of locking affects all types of modifications, it is discussed in a separate section later in this chapter.

Other, rarer types of error 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 of any kind, you can tell the database server to put the database back the way it was. Here is the preceding example extended to use transactions:

```
$ begin work;                          /* start the transaction*/
$ 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, */
$ rollback work;                   /* ...put everything back */
else                               /* everything worked OK, */
$ commit work;                     /* ...finish transaction */
```

An important point in this example is that the program saves the important return values in the SQLCA before it ends the transaction. The reason is that both the ROLLBACK WORK and COMMIT WORK statements, like all SQL statements, set return codes in the SQLCA. 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 no matter what goes wrong, the database is left in a known, predictable state. There is never a question about how much of the modification is completed; either all of it is, or none of it is.

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 shown in the **INFORMIX-4GL** example in Figure 7-1.

```
WHENEVER ERROR CONTINUE{do not terminate on error}
BEGIN WORK  {start transaction}
DELETE FROM items
    WHERE order_num = o_num
IF (status >= 0) THEN{no error on first delete}
    DELETE FROM orders
        WHERE order_num = o_num
END IF
IF (status >= 0) THEN{no error on either delete}
    COMMIT WORK
ELSE        {problem on some delete}
    DISPLAY "Error ", status, " deleting."
    ROLLBACK WORK
END IF
```

Figure 7-1   A fragment of 4GL that deletes from two tables

The logic of this program is much the same whether or not transactions are used. But if they are not used, the person who saw the error message has a much more difficult set of decisions to make. Depending on when the error occurred, the situation is as follows:

- No deletions were performed; all rows with this order number remain in the database.

- Some item rows were deleted but not all; 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. The user 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 through a cursor to delete the row that was last fetched. In this way, you can program deletions based on conditions that cannot be tested in a WHERE clause. An example appears in Figure 7-2.

```
int delDupOrder()
{
$    int ord_num;
     int dup_cnt, ret_code;
$    declare scan_ord cursor for
         select order_num, order_date
             into $ord_num
             from orders for update;
$    open scan_ord;
     if (sqlca.sqlcode != 0) return (sqlca.sqlcode);
$    begin work;
     for(;;)
     {
$        fetch next scan_ord;
         if (sqlca.sqlcode != 0) break;
         dup_cnt = 0; /* default in case of error */
$        select count(*) into dup_cnt from orders
             where order_num = $ord_num;
         if (dup_cnt > 1)
         {
$            delete where current of scan_ord;
             if (sqlca.sqlcode != 0) break;
         }
     }
     ret_code = sqlca.sqlcode;
     if (ret_code == 100)/* merely end of data */
$    commit work;
     else    /* error on fetch or on delete */
$    rollback work;
     return (ret_code);
}
```

Figure 7-2   An unsafe* ESQL/C *function that deletes through a cursor (see Note)

Note: The design of the ESQL/C *function in Figure 7-2 is unsafe. It depends for correct operation on the current "isolation level" (which is discussed later in the chapter). Even when it works the way it is meant to work, its effects depend on the physical order of rows in the table, which is not generally a good idea.*

The purpose of the function is to delete rows containing 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 be used to 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 is the step that fails without the correct isolation level, as described in a later section).

In the demonstration database, with its unique index on this table, the count returned to **dup_cnt** is always 1. 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 one deletes all duplicate rows except the last one delivered by the database server. That ordering has nothing to do with the contents of the rows or their meanings. You might think of improving the function in Figure 7-2 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 sketched later in the chapter.

Using INSERT

You can embed the INSERT statement in programs. Its form and use in a program are the same as described in Chapter 5 of this manual, 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 using a cursor.

Using an Insert Cursor

The DECLARE CURSOR statement has many variations. Most of them are used to create cursors for different kinds of scans over data, but one variation creates a special kind of cursor, an *insert cursor*. You use an insert cursor with the PUT and FLUSH statements to efficiently insert rows into a table in bulk.

Declaring an Insert Cursor

You create an insert cursor by declaring a cursor to be FOR an INSERT state-
ment instead of a SELECT statement. You cannot use such a cursor to fetch
rows of data; you can only use it to insert them. An example of the declara-
tion of an insert cursor is shown in Figure 7-3.

```
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)
```

Figure 7-3   *A 4GL fragment that declares an insert cursor*

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 data-
base server, and it lets the database server insert the rows with less difficulty.
As a result, the insertions go faster.

The minimum size of the insert buffer is set for any implementation of
embedded SQL; you have no control over it (it is typically one or two kilo-
bytes). The buffer is always made large enough to hold at least two rows of
inserted values. It is large enough to hold multiple rows when the rows are
shorter than the minimum buffer size.

Inserting with a Cursor

The code in Figure 7-3 prepares an insert cursor for use. The continuation shown in Figure 7-4 shows 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.

```
WHENEVER ERROR CONTINUE {do not terminate on error}
BEGIN WORK
OPEN new_custs
WHILE status = 0
    CALL next_cust() RETURNING the_company,the_fname,the_lname
    IF the_company IS NULL THEN
        EXIT WHILE
    END IF
    PUT new_custs
END WHILE
IF status = 0 THEN      {no problem in a PUT}
    FLUSH new_custs      {write any last rows}
END IF
IF status = 0 THEN      {no problem writing}
    COMMIT WORK          {..make it permanent}
ELSE
    ROLLBACK WORK        {retract any changes}
END IF
```

Figure 7-4   *Continuing Figure 7-3, the code that uses the insert cursor*

The code in Figure 7-4 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 up, 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 is used to write any rows that remain in the buffer, after which the transaction is terminated.

Examine the INSERT statement in Figure 7-3 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 Figure 7-4 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 was 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. An error can occur in this case.

The values returned into the SQL Communications Area give the program the information it needs to sort out all these cases. SQLCODE is set after every PUT statement—to zero if there is no error and to a negative error code if there is an error.

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 were 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 (Figure 7-4) once again to see how SQLCODE is used. 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 is rolled 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 is rolled back. This can only occur 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 all. Then the SQL status is zero, and the FLUSH operation is performed. If the FLUSH operation produces an error code, the transaction is rolled 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. This is the case in which all of the values listed in the INSERT statement are constants—no expressions and no host variables, just literal numbers and strings of characters. No matter how many times such an INSERT operation is performed, the rows it produces are identical. In that case, there is no point in copying, buffering, and transmitting each identical row.

Instead, for this kind of INSERT operation, the PUT statement does nothing whatever except to increment a counter. When finally a FLUSH operation is 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.

It is not common to insert a quantity of identical rows. You can do it when you first establish a database, to populate a large table with null data.

An Insert Example

The preceding section on the DELETE statement contains an example whose purpose is to look for and delete duplicate rows of a table (see "Deleting with a Cursor" on page 7-7). A better way to do the same thing is to select the desired rows, instead of deleting the undesired ones. The **INFORMIX-4GL** code in Figure 7-5 shows one way to do this. The example is written in **INFORMIX-4GL** to take advantage of some features that make SQL programming easy.

```
BEGIN WORK
INSERT INTO new_orders
    SELECT * FROM ORDERS main
        WHERE 1 = (SELECT COUNT(*) FROM ORDERS minor
                      WHERE main.order_num = minor.order_num)
COMMIT WORK

DEFINE ord_row RECORD LIKE orders,
    last_ord LIKE orders.order_num
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
DECLARE ins_row CURSOR FOR
    INSERT INTO new_orders VALUES (ord_row.*)

BEGIN WORK
OPEN ins_row
LET last_ord = -1
FOREACH dup_row
    IF ord_row.order_num <> last_ord THEN
        PUT ins_row
        LET last_ord = ord_row.order_num
    END IF
END FOREACH
CLOSE ins_row
COMMIT WORK
```

Figure 7-5   A 4GL program that re-creates a table without duplicates

This example begins with an ordinary INSERT statement that finds all the nonduplicated rows of the table and inserts them into another table, presumably created before the program started. That leaves only the duplicate rows. (Remember, in the demonstration database the **orders** table has a unique index and cannot have duplicate rows. This example deals with some other database.)

In **INFORMIX-4GL**, you can define a data structure *like* a table; the structure is automatically given one element for each column in the table. The **ord_row** structure is a buffer to hold one row of the table.

The code in Figure 7-5 then declares two cursors. The first, called **dup_row**, returns the duplicate rows in the table. Because it is only for input, it can use the ORDER BY clause to impose some order on the duplicates other than the physical record order used in Figure 7-2 on page 7-7. In this example, the duplicate rows are ordered by their dates (the oldest one is kept), but you can use any other order based on the data.

The second cursor is an insert cursor. It is written to take advantage of the *asterisk* notation of **INFORMIX-4GL**; you can supply values for all columns simply by naming a record, with an asterisk to indicate *all fields*.

The remainder of the code examines the rows returned through **dup_row**. It inserts the first one from each group of duplicates into the new table, and disregards the rest.

This example uses the simplest kind of error handling. Unless told otherwise, an **INFORMIX-4GL** program automatically terminates when an error code is set in SQLCODE. In that event, the active transaction is also rolled back. This program relies on that behavior; it assumes that if it reaches the end there were no errors and the transaction can be committed. This kind of error handling is acceptable when errors are unlikely and the program is used by people who do not need to know why the program terminates.

Using UPDATE

You can embed the UPDATE statement in a program in any of the forms described in Chapter 5, 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 addressed by a cursor.

How Many Rows Were Affected? SQLCODE and SQLERRD

When your program uses a cursor to select rows, it can test SQLCODE for 100, the end-of-data return code. This is set to indicate that no rows, or no more rows, satisfy the query conditions. The end-of-data code is set in SQLCODE only following SELECT statements; it is not used following DELETE, INSERT, or UPDATE statements.

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 code even when the source of the inserted rows is a SELECT statement and it selected no rows. The INSERT statement is a success since it inserted as many rows as it was asked to do (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 an Update Cursor

An *update cursor* permits you to delete or update the current row; that is, the most recently fetched row. Here is an example (in **INFORMIX-ESQL/COBOL**) of the declaration of an update cursor:

```
EXEC SQL
    DECLARE names CURSOR FOR
        SELECT fname, lname, company
        FROM customer
    FOR UPDATE
END-EXEC
```

The program that uses this cursor can fetch rows in the usual way.

```
EXEC SQL
    FETCH names INTO :FNAME, :LNAME, :COMPANY
END-EXEC.
```

If the program then decides that the row needs to be changed, it can do so.

```
IF COMPANY IS EQUAL TO "SONY"
    EXEC SQL
        UPDATE customer
            SET fname = "Midori", lname = "Tokugawa"
            WHERE CURRENT OF names
    END-EXEC.
```

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 required nevertheless.

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 fetching a row for a cursor that is not declared with that keyword. This 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 later in this chapter.)

Updating Specific Columns

You can declare a cursor to update specific columns. Here is a revision of the preceding example that uses this feature:

```
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 takes place. An attempt to delete using WHERE CURRENT OF is also rejected, since deletion affects all columns.

UPDATE Keyword Not Always Needed

The ANSI standard for SQL does not provide for the clause FOR UPDATE in a cursor definition. When a program uses an ANSI-compliant database, it can update or delete using any cursor.

Cleaning up a Table

A final, hypothetical example of using 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.

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 (using the

ALTER TABLE command). This column is to have unique integer values installed. Figure 7-6 sketches the **INFORMIX-ESQL/C** code to accomplish these things.

```
$ char[80] dcol;
$ short int dcolint;
$ int sequence;
$ declare target_row cursor for
    select datcol
        into $dcol:dcolint
        from target
    for update of serials;
$ begin work;
$ open target_row;
if (sqlca.sqlcode == 0) $ fetch next target_row;
for(sequence = 1; sqlca.sqlcode == 0; ++sequence)
{
    if (dcolint < 0) /* null datcol */
        $ delete where current of target_row;
    else
        $ update target set serials = $sequence
            where current of target_row;
}
if (sqlca.sqlcode >= 0) $ commit work;
else $ rollback work;
```

Figure 7-6   Cleaning up a handmade table using an update cursor

Concurrency and Locking

If your database is contained in a single-user workstation, with no network connecting it to other machines, your programs can modify data freely. But 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 is *concurrency*: two or more independent uses of the same data at the same time.

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

Just the same, unless there are controls 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.

The database server prevents errors of this kind by imposing 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

Since a lock serializes access to one piece of data, it reduces concurrency; any other programs that want access to that piece of data must wait. The database server can place a lock on a single row, on a disk page (which holds multiple rows), on a whole table, or on 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 objects, the greater concurrency (and performance) can be.

This section discusses how a program can achieve two goals:

- To place all of the locks that are needed to ensure data integrity
- To 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. Here is an example. Suppose that your program is fetching rows through the following cursor:

```
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. Suppose that, at about the same time your program fetches the rows produced by that query, another user's program executes this update:

```
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. There are four possibilities concerning 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 your program has yet to read. Then it executes COMMIT WORK.

    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 ROLLBACK WORK.

    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 case 1, the update is complete before your query begins. It makes no difference whether it finished a microsecond ago or a week ago.

In case 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 second pair of chances, however, can be very important to the design of some applications. In case 3, the query returns a mix of updated and original data. That can be a negative thing in some applications. In others, for instance one that is taking an average of all prices, it might not matter at all.

In case 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 for the following reasons:

- Your program fetches the row.
- Another program updates or deletes the row.
- Your program updates or deletes WHERE CURRENT OF *names*.

You control concurrent events like these using the locking and *isolation level* features of the database server.

How Locks Work

The **INFORMIX-OnLine** database server supports a complex, flexible set of locking features that is described in this section. Other Informix database servers have simpler locking systems.

Kinds of Locks

INFORMIX-OnLine supports three kinds of locks that it uses in different situations:

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. It is used when the program intends to change the object.

An exclusive lock cannot be placed where any other kind of lock exists. Once one 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 lock exists; however, once it has been placed, other shared locks can join it. Later, it can be changed to an exclusive lock.

Lock Scope

You can apply locks to entire databases, to entire tables, to disk pages, to single rows, or to index-key values. The size of the object being locked is referred to as the *scope* of the lock (sometimes 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 statements DATABASE or CREATE DATABASE. 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

The 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 only released when the database is closed. That can be done explicitly with the CLOSE DATABASE statement, or implicitly by executing another DATABASE statement.

Since locking a database reduces concurrency in that database to zero, it makes programming very simple—concurrent effects cannot happen. However, you should only lock a database 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 is done automatically. **INFORMIX-OnLine** always locks an entire table while it performs any of the following statements:

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- RENAME COLUMN
- RENAME TABLE

The completion of the statement (or end of the transaction) releases the lock. An entire table also can be locked automatically during certain queries, as described later in this section.

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. **INFORMIX-OnLine** achieves the same degree of protection by setting the isolation level, as described in the next section; this allows greater concurrency. However, all Informix database servers support the LOCK TABLE statement.

An exclusive table lock prevents any concurrent use of the table. This has 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.

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.

INFORMIX-OnLine stores data in units called *disk pages* (its disk-storage methods are described in detail in Chapter 10 of this manual). 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.

The choice between locking by rows or locking by pages is established for a table when it is created. **INFORMIX-OnLine** supports a clause, LOCK MODE, to specify either page or row locking. You can specify lock mode in the CREATE TABLE statement and later change it with ALTER TABLE. (Other Informix database servers do not offer the choice; they lock by page or by row, whichever makes the better implementation.)

Page and row locking are used identically. Whenever **INFORMIX-OnLine** 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 it uses page locking, a key lock is placed on the index page that contains the key, or that would contain the key if the key existed.

The Duration of a Lock

The program controls the duration of a database lock. A database lock remains until it is released by the closing of the database.

The duration of a table lock depends on whether the database uses transactions. If it does not (that is, if there is no transaction log and the COMMIT WORK statement is not used), 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 are used, and on whether transactions are in use.

When transactions are used, the end of a transaction releases all table, row, page, and index locks. This is an important point: 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 succeeds, the database server knows that no other program can alter that row. Since 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 it 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 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. This prevents the occurrence of the following error:

- Program *A* deletes a row.
- Program *B* inserts a row having 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 *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 reading are controlled by the current isolation level, which is the next topic.

Setting the Isolation Level

The *isolation level* is the degree to which your program is isolated from the concurrent actions of other programs. **INFORMIX-OnLine** offers a choice of four isolation levels. It implements them by setting different rules for how a program uses locks when it is reading.

You set the isolation level using the command SET ISOLATION LEVEL. Not all database servers support this command, so before executing it in a program, test to see whether the database server supports it.

The simplest test is to execute the statement and either ignore the return code, or test the return code and use it as an indication of which database server is in use. (Error code -513 displays the message `Statement not available with this database engine`.)

Dirty Read Isolation

The simplest isolation level, Dirty Read, amounts to virtually no isolation at all. 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 Dirty Read isolation, a program never sees a row in which some columns have been updated and some not. However, a program using 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 (case 4 in the list of concurrency issues on page 7-19).

Dirty Read is the most efficient 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 database is held in an exclusive lock.
- It is certain that only one program is using the database.

Committed Read Isolation

When a program requests the Committed Read isolation level, **INFOR-MIX-OnLine** guarantees that it never returns a row that is not committed to the database. This prevents the situation in case 4 in the list of concurrency issues on page 7-18 (reading data that is not committed, and which is subsequently rolled back).

Committed Read is implemented very simply. Before fetching a row, the database server tests to determine whether an updating process placed a lock on the row. If not, it returns the row. Since rows that are updated but not committed have locks on them, this test ensures that the program does not read uncommitted data.

Committed Read does not actually place a lock on the fetched row; hence, it is almost as efficient as Dirty Read. It is appropriate for use when each row of data is processed as an independent unit, without reference to other rows in the same or other tables.

Cursor Stability Isolation

The next level is called Cursor Stability. When it is in effect, the database 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).

Cursor Stability ensures that a row does not change while the program examines it. This is important when the program updates some other table based on the data it reads from this row. Because of Cursor Stability, the program is assured that the update is based on current information. It prevents the use of *stale data*.

Here is an example to illustrate this point. In terms of the demonstration database, Program *A* wants to insert a new stock item for manufacturer Hero. Concurrently, program *B* wants to delete manufacturer Hero and all stock associated with it. The following sequence of events can take place:

1. Program *A*, operating under Cursor Stability, fetches the Hero row of the **manufact** table to learn the manufacturer code. This 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 have happened instead:

1. Program *A* reads the Hero 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 Hero row is now invalid, inserts a new row of **stock** using the manufacturer code HRO.

6. Program *A* ends.

At the end, there is a row in **stock** that has no matching manufacturer code in **manufact**. Furthermore, Program *B* apparently has a bug; it did not delete the rows it was supposed to delete. The use of the Cursor Stability isolation level prevents these effects.

(It is possible to rearrange the preceding scenario so that it fails 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, it is essential that all programs involved use the same sequence of access.)

Since Cursor Stability locks only one row at a time, it restricts concurrency less than does a table lock or database lock.

Repeatable Read Isolation

The Repeatable-Read isolation level asks the database server to put a lock on every row the program fetches. The locks placed are shareable for an ordinary cursor and promotable for an update cursor. The locks are placed one at a time as rows are fetched. They are not released until the cursor is closed 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 6 of this manual). 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, you must 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 only a fixed number of locks for all the programs it serves. If that table fills up, the database server cannot place a lock and returns an error code. The person who administers an **INFOR**-**MIX**-**OnLine** system can monitor the lock table and tell you when it is heavily used.

Repeatable Read is automatically the isolation level in an ANSI-compliant database. Repeatable Read is required to ensure operations behave in accordance with the ANSI standard for SQL.

Setting the Lock Mode

The lock mode determines what happens when your program encounters locked data. One of three things occurs when a program attempts to fetch or modify a locked row:

- The program receives an immediate return from the database server with an error code in SQLCODE.
- The program is suspended until the lock is removed by the program that placed it.
- The program is suspended for a time and then, if the lock is not removed, it receives an error-return code from the database server.

You choose among these results with the command SET LOCK MODE.

Waiting for Locks

If you prefer to wait (and this is the best choice for many applications), execute the following command:

```
SET LOCK MODE TO WAIT
```

When this lock mode is set, your program usually ignores the existence of other, concurrent programs. When it needs to access a row that another program has locked, your program 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 very long (although properly designed applications should hold their locks very briefly). When the possibility of a long delay is not acceptable, a program can execute the following command:

```
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. It is up to the program to roll back its current transaction and try again.

Not waiting is the initial setting 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 commands the program executes.

Waiting a Limited Time

When you use **INFORMIX**-**OnLine**, you have an additional choice: you can ask the database server to set an upper limit on a wait. You can issue the following command:

```
SET LOCK MODE TO WAIT 17
```

This 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 each other's progress. Each program has a lock on some object that the other program wants to access. A deadlock only arises when all programs concerned set their lock modes to wait for locks.

INFORMIX-**OnLine** 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. Thus, 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 at different database servers. In this case, **INFORMIX**-**OnLine** cannot instantly detect the deadlock. (Perfect deadlock detection requires excessive communications traffic between 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 maximum is set for the database server and can be changed by the person who installs it.

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 commands when it is starting 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.

Locking with Other Database Servers

INFORMIX-OnLine manages its own locking so that it can provide the different kinds of locks and levels of isolation described in the preceding topics. Other Informix database servers implement locks using the facilities of the host operating system, and cannot provide the same conveniences.

Some host operating systems provide locking functions as operating system services. In these systems, database servers support the SET LOCK MODE statement.

Some host operating systems do not provide *kernel-locking* facilities. In these systems, the database server performs its own locking based on small files that it creates in the database directory. (These files have the suffix **.lok**.)

You can tell in which kind of system your database server is running by executing the SET LOCK MODE statement and testing the error code, as in the following fragment of **INFORMIX-ESQL/C** code:

```
#define LOCK_ONLINE 1
#define LOCK_KERNEL 2
#define LOCK_FILES 3
int which_locks()
{
    int locktype;
    locktype = LOCK_FILES;
$   set lock mode to wait 30;
    if (sqlca.sqlcode == 0) locktype = LOCK_ONLINE;
    else
    {
$       set lock mode to wait;
        if (sqlca.sqlcode == 0) locktype = LOCK_KERNEL;
    }
$   set lock mode to not wait; /* restore default condition */
    return(locktype);
}
```

If the database server does not support the SET LOCK MODE statement, your program is effectively always in NOT WAIT mode; that is, whenever it tries to lock a row locked by another program, it receives an error code immediately.

Isolation While Reading

Informix database servers other than **INFORMIX-OnLine** do not normally place locks when fetching rows. There is nothing comparable to the shared locks used by **INFORMIX-OnLine** to implement the Cursor Stability isolation level.

If your program fetches a row with a singleton SELECT statement or through a cursor that is not declared FOR UPDATE, the row is fetched immediately, regardless of whether it is locked or modified by an unfinished transaction.

This design produces the best performance (especially when locks are implemented by writing notes in disk files), but you must be aware that the program can read rows modified by uncommitted transactions.

You can obtain the effect of Cursor Stability isolation by declaring a cursor FOR UPDATE, and then using it for input. Whenever the database server fetches a row through an update cursor, it places a lock on the fetched row. (If the row is locked already, the program waits or receives an error, depending on the lock mode.) When the program fetches another row without updating the current one, the lock on the current row is released and the new row is locked.

Thus, by fetching through an update cursor, you can be sure that the fetched row is locked as long as you are using it. (The row cannot become *stale*.) You are also assured of fetching only committed data, since locks on rows that are updated are held until the end of the transaction. Depending on the host operating system and the database server, you might experience a performance penalty for using an update cursor this way.

Locking Updated Rows

When a cursor is declared FOR UPDATE, locks are handled as follows: Before a row is fetched, it is locked; or if it cannot be locked, the program is made to wait or an error is returned.

The next time a fetch is requested, the database server notes whether the current row is modified (using either UPDATE or DELETE with WHERE CURRENT OF), and whether a transaction is in progress. If both of these things are true, the lock on the row is retained. Otherwise, the lock is released.

Thus, if you perform updates within a transaction, all updated rows remain locked until the transaction ends. Rows that are not updated are locked only while they are current. Rows updated outside a transaction, or in a database that does not use transaction logging, are also unlocked as soon as another row is fetched.

Hold Cursors

When transaction logging is used, the database server guarantees that any-
thing done within a transaction can be rolled back at the end of it. To do this
reliably, the database server normally applies the following rules:

- All cursors are closed by the ending of a transaction.
- All locks are released by the ending of a transaction.

These rules are normal with all database systems that support transactions,
and for most applications they do not cause any trouble. However, there is a
program design that is possible when transactions are not used that becomes
impractical when transactions are added. The design is sketched in
pseudocode in Figure 7-7.

```
DECLARE master CURSOR FOR ...
DECLARE detail CURSOR FOR ... FOR UPDATE
OPEN master
LOOP:
    FETCH master INTO ...
    IF (the fetched data is appropriate) THEN
        BEGIN WORK
        OPEN detail USING data read from master
        FETCH detail ...
        UPDATE ... WHERE CURRENT OF detail
        COMMIT WORK
    END IF
END LOOP
CLOSE MASTER
```

Figure 7-7   ***A pseudocode sketch of one common form of database application***

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 in the sketch in Figure 7-7), the
COMMIT WORK statement following the UPDATE closes all cursors—includ-
ing the master cursor.

The simplest alternative is to move the COMMIT WORK and BEGIN WORK
statements to be the last and first ones, respectively, so that the entire scan
over the master table is one large transaction. This is sometimes possible, but
it may become impractical if there are very many rows to update. The num-
ber of locks can be too large, and they are held for the duration of the
program.

A solution supported by Informix database servers 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 also that you understand the programs that are running concurrently. The reason is that whenever COMMIT WORK is executed, all locks are released, including any locks placed on rows fetched through the hold cursor.

This has little importance if the cursor is used as intended, for a single forward scan over a table. However, you are allowed to 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

A program can execute the INSERT, DELETE, and UPDATE statements just as they were described in Chapter 5. A program also can scan through a table with a cursor, updating or deleting selected rows. It also can use a cursor to insert rows, with the benefit that the rows are buffered and sent to the database server in blocks.

In all of these activities, the program must take pains to detect errors and to return the database to a known state when one occurs. The most important tool for doing this is the transaction. Without transaction logging, the program has a more difficult time recovering from errors.

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.

Building a Data Model

8

Chapter Overview

The first step in creating a database is to construct a data model—a precise, complete definition of the data to be stored. This chapter contains a cursory overview of one method of doing this. The chapters that follow describe how to implement a data model once you design it.

Why Build a Data Model

A *data model* is a precise, complete definition of the data you plan to store. You probably already have an intuitive model of the data in mind, but you should complete it using some type of formal notation. This helps your design in two ways:

- It makes you think through the data model completely.

  A mental model often contains unexamined assumptions; formalizing the design reveals these points.

- It is easier to communicate your design to other people.

  A formal statement makes the model explicit, so that others can return comments and suggestions in the same form.

Extended Relational Analysis

Different books present different formal methods of modeling data. Most methods force you to be thorough and precise. If you have already learned some method, by all means use it.

This chapter presents a summary of *Extended Relational Analysis,* a modeling method devised by Relational Systems Corporation and taught in training courses presented by Informix and other companies. This modeling method is carried out in three steps:

1. Identify the entities (fundamental objects) that the database describes.
2. Identify the relationships between the entities.
3. Identify the attributes (intrinsic features) associated with the entities and relationships.

Chapter 9 adds a fourth step, specifying the domain (data type) of each attribute.

The end product of modeling is a set of tables like those in Figure 8-1, which shows the final set of tables for a personal address book. The personal address book is an example developed in this chapter. It is used rather than the **stores5** database used in the rest of this book because it is small enough to be developed completely in one chapter, but large enough to show the entire method.

Name

NameString	AddressCode	BirthDate
PK UA	FK	
Gaius C. Caesar SPQR, S.A.	1001 1020	17-03-1990 (null)

Address

AddressCode	Street	City	StaProv	Postcode
PK SA				
1001 1020	2430 Tasso St. #8 411 Bohannon Drive	St. Louis Lenaxa	CA Abta.	83267-1048 TW2 5AQ

Voice

VceNumber	VceType
PK UA	
1-800-274-8184 011-49-89-922-030	home car

Fax

FaxNumber	AddressCode	VceNumber	OperFrom	OperTill
PK UA	FK	FK		
926-6741 61-62-435-143	1001 1020	926-6300 (null)	0800 1730	0400 1900

NameFax

NameString	FaxNumber
PK FK	PK FK
Gaius C. Caesar SPQR, S.A.	926-6741 61-62-435-143

NameVce

NameString	VceNumber
PK FK	PK FK
Gaius C. Caesar SPQR, S.A.	1-800-274-8184 011-49-89-922-030

Modem

MdmNumber	NameString	VceNumber	AddressCode	B300	B1200	B2400
PK UA	FK NN	FK	FK			
416-566-7024 33 1 42 70 67 74	Nite Owl BBS CompuServe	926-6300 (null)	1001 1020	Y Y	Y Y	N Y

Figure 8-1   The data model of a personal address book

Basic Ideas

The following ideas are fundamental to most relational data model methods, including Extended Relational Analysis.

Tables, Rows, and Columns

You are already familiar with the idea of a *table* composed of *rows* and *columns.* But you must respect four rules while you are defining the tables of a formal data model:

- Rows must stand alone.

  Each row of a table is independent and does not depend on any other row of the *same* table. As a consequence, the order of the rows in a table is not significant in the model. The model should still be correct even if all the rows of a table are shuffled into random order.

  After the database is implemented, you can tell the database server to store rows in a certain order for the sake of efficiency, but that does not affect the model.

- Rows must be unique.

  In every row, some column must contain a unique value. If no single column has this property, the values of some group of columns taken as a whole must be different in every row.

- Columns must stand alone.

  The order of columns within a table has no meaning in the model. The model should still be correct even if the columns are rearranged.

  After the database is implemented, programs and stored queries that use an asterisk to mean *all columns* depends on the final order of columns, but that does not affect the model.

- Column values must be unitary.

  A column can contain only single values, never lists or repeating groups. Composite values must be broken into separate columns. For example, if at some point you treat a person's first and last names as separate values, they must be in separate columns, not in a single *name* column.

If your previous experience is only with data organized as arrays or sequential files, these rules might seem unnatural. However, relational database theory shows that you can represent all types of data using only tables, rows, and columns that follow these rules. With only a little practice, the rules become automatic.

Primary Keys

The *primary key* of a table is the column whose values are different in every row. Because they are different, they make each row unique. If there is no one such column, the primary key is a composite of two or more columns whose values, taken together, are different in every row.

Every table in the model must have a primary key. This follows automatically from the rule that all rows must be unique. If necessary, the primary key is composed of all the columns taken together.

Null values are never allowed in a primary key column. Null values are not comparable; that is, they cannot be said to be alike or different. Hence, they cannot make a row unique from other rows. If a column permits null values, it cannot be part of a primary key.

In a diagram, the primary key columns of a table are flagged with the note PK, as in this table from Figure 8-1:

```
Modem
MdmNumber
  PK UA
416-566-7024
33 1 42 70 67 74
```

The note PK means *primary key.* The note UA means *user assigned.* That is, the values in this primary key come from the real world; they are not codes assigned by the system. (A system-assigned key is flagged SA.)

Candidate Keys

Sometimes more than one column or group of columns qualifies as the primary key. Such columns or groups are called *candidate keys.* Choose the candidate with the fewest columns to be the primary key of the table.

All candidate keys are worth noting because their property of uniqueness makes them predictable in a SELECT operation. When you select the columns of a candidate key, you know the result can contain no duplicate rows. Sometimes that allows you to predict how many rows are returned. It always means that the result can be a table in its own right, with the selected candidate key as its primary key.

It is optional to flag candidate-key columns as CK in table diagrams.

Foreign Keys (Join Columns)

A *foreign key* is simply a column or group of columns in one table that contain values that match the primary key in another table. Foreign keys are used to join tables; in fact, most of the *join columns* referred to earlier in this book are foreign-key columns. Foreign-key columns are flagged FK in the model tables, as in this portion of Figure 8-1:

Fax

FaxNumber	AddressCode	VceNumber
PK UA	FK	FK
926-6741	1001	926-6300
61-62-435-143	1020	(null)

Foreign keys are noted wherever they appear in the model because their presence restricts your ability to delete rows from tables. Before you can delete a row safely, you should delete all rows that refer to it through foreign keys. Otherwise, the database is in an inconsistent state; some rows contain foreign key values that refer to nonexistent data.

When such an inconsistency occurs, that is, when a foreign key value in one table does not have a match in any primary key, *referential integrity* has been violated. You can always preserve referential integrity by deleting all foreign-key rows before you delete the primary key to which they refer. If you are imposing referential constraints on your database, the database itself does not permit you to delete primary keys with matching foreign keys. Nor does it permit you to add a foreign-key value that does not reference an existing primary-key value. Using referential constraints when implementing your model is covered in Chapter 9.

Step 1: Name the Entities

An *entity* is a type of person, place, or thing to be recorded in the database. If the data model were a language, entities would be its nouns.

The first step in modeling is to choose the entities to record. Each one becomes a table in the model.

You probably can list several entities immediately. However, if other people use the database, you should poll them for their understanding of what kinds of fundamental *things* the database should contain.

When the list of entities seems complete, prune it by making sure that each one has the following qualities:

- It is significant.

    List only entities that are important to the users of the database and worth the trouble and expense of computer tabulation.

- It is generic.

    List only types of things, not individual instances. For instance, *symphony* might be an entity, but *Beethoven's Fifth* would be an instance.

- It is fundamental.

    List only entities that exist independently, without needing something else to explain them. Anything you could call a trait, a feature, or a description is merely an attribute, not an entity. For example, a *part number* cannot exist without a more fundamental *part* entity. Also, do not list things that you can derive from other entities; avoid, for example, any sum, average, or other quantity that you can calculate in a SELECT expression.

- It is unitary.

    Be sure that each entity you name represents a single class—that it cannot be broken down into subcategories each with its own features. In planning the address book model (see "The Address-Book Example" on page 8-11) an apparently simple entity, the telephone number, turns out to consist of three categories each with different features.

These choices are neither simple nor automatic. To discover the best choice of entities, you must think deeply about the nature of the data you want to store. Of course, that is exactly the point of making a formal data model.

Entity Keys

Each entity you choose is represented as a table in the model. The table stands for the entity as an abstract concept, while each row represents a specific, individual *instance* of the entity. For example, if *customer* is an entity, a **customer** table represents the idea of customer; in it, each row represents one specific customer.

An entity is a table and every table must have a primary key; therefore, your next task is to specify a primary key for each entity. That is, you must designate some quantifiable characteristic of the entity that distinguishes each instance from every other.

User-Assigned Keys

Some entities have ready-made primary keys, such as catalog codes or identity numbers, defined outside the model. These are user-assigned keys, flagged UA in the model.

Composite Keys

Other entities lack features that are reliably unique. Different people can have identical names; different books can have identical titles. You can usually find a composite of features that work (it is rare for people to have identical names and identical addresses, or for different books to have identical titles, authors, and publication dates).

System-Assigned Keys

A system-assigned primary key is usually preferable to a composite key. A system-assigned key is a number or code that is attached to each instance of an entity when it is first entered into the database. The easiest system-assigned keys to implement are serial numbers because the database server can generate them automatically. However, the people who use the database might not like a plain numeric code. Other codes can be based on actual data; for example, an employee identification code can be based on the person's initials, combined with the digits of the date they were hired.

Time-Dependent Keys

An entity key can be naturally composite and it can include foreign keys. Both types of keys are often required when the data involves time. For example, a frequently crucial entity is *billable time*, a span of time that can be charged to a client. A law office or a marriage counselor might use entities such as *advisor* and *client*; when advisor meets client, an instance of billable time occurs.

A billable-time entity needs at least three things to make it unique:

- The identity of the advisor (a foreign key referring to the *advisor* entity)
- The date
- The start time of the service

Entity Tables

After deciding on the primary keys, record your decisions as diagrams. For each entity, draw the corner of a table as shown in Figure 8-2, noting the name of the entity and the chosen primary key columns.
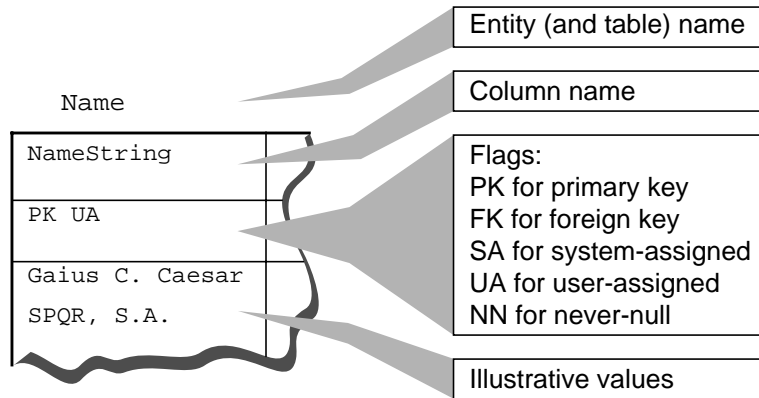
Figure 8-2   *Diagram of an entity table*

The Address-Book Example

Suppose that you create a database that computerizes a personal address book. The database model must record the names, addresses, and telephone numbers of people and organizations that its user deals with in the course of business.

The first step is to define the entities, and the first thing you might do is look carefully at a page from an address book to see what entities are there (see Figure 8-3).
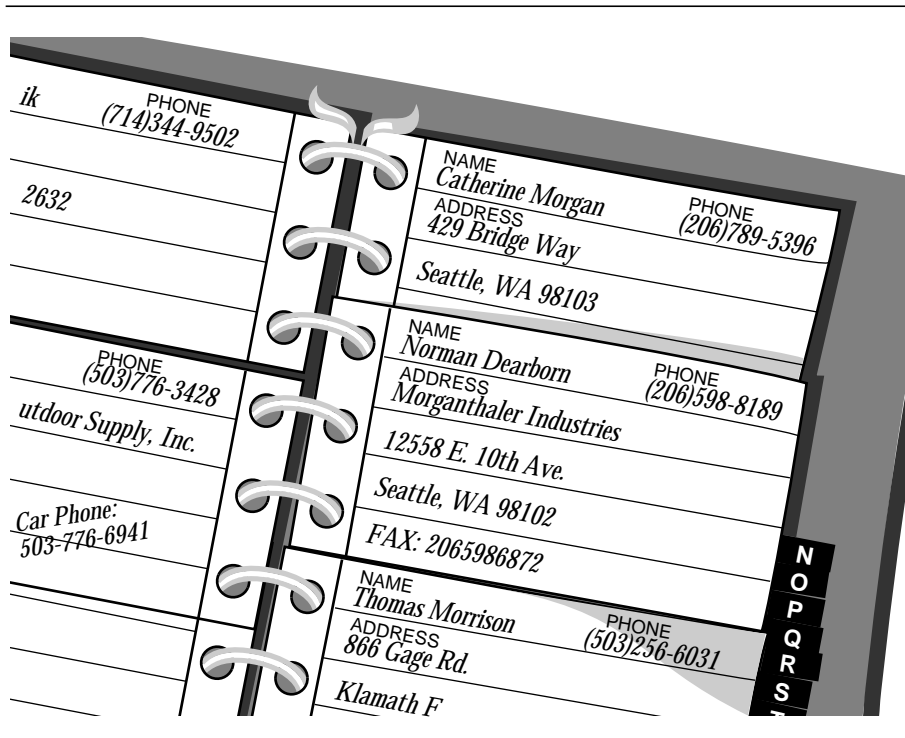
Figure 8-3   Part of a page from an address book

The physical form of the existing data can be misleading. Do not let the lay-out of pages and entries in the address book mislead you into trying to spec-ify an entity that represents one entry in the book—some kind of alphabetized record with fields for name, number, and address. Remember it is not the medium you want to model, it is the data.

At first glance, the fundamental data entities recorded in an address book include

- Names (of persons and organizations)
- Addresses
- Telephone numbers

Do these data entities meet the criteria given earlier? They are clearly signif-icant to the model and generic.

Are they fundamental? A good test is to ask if an entity can vary in number independently of any other entity. After thinking about it, you realize that an address book sometimes lists people who have no number or current address

(people who move or change jobs). An address book also can list both addresses and numbers that are used by more than one person. All three of these entities can vary in number independently; that strongly suggests they are fundamental, not dependent.

Are they unitary? Names can be split into personal names and corporate names. After thinking about it, you decide that all names should have the same features in this model; that is, you do not plan to record different information about a company than you would about a person. Likewise, you decide there is only one kind of address; there is no need to treat home addresses differently from business ones.

However, you also realize that there is not one kind of telephone number, but three. There are *voice* numbers that are answered by a person, *fax* numbers that connect to a fax machine, and *modem* numbers that connect to a computer. You decide that you want to record different information about each kind of number, so these three are different entities.

After you choose primary keys, the diagram of the tables of the model might look like Figure 8-4.

Name

NameString
PK UA
Gaius C. Caesar SPQR, S.A.

Address

AddressCode
PK SA
1001 1020

Voice

VceNumber
PK UA
1-800-274-8184 011-49-89-922-030

Fax

FaxNumber
PK UA
926-6741 61-62-435-143

Modem

MdmNumber
PK UA
416-566-7024 33 1 42 70 67 74
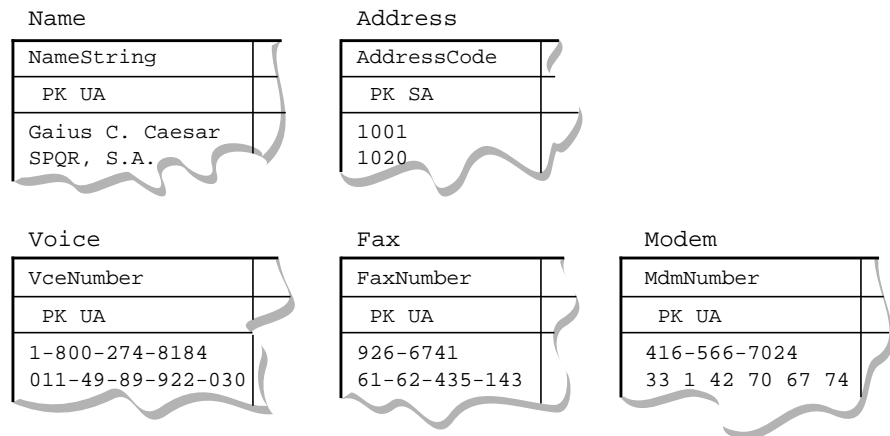
Figure 8-4   *The tables of an address-book model after primary keys are chosen*

This diagram reflects some important decisions. First, the name of a person or organization is being used as a primary key (user-assigned). That would not be acceptable if two organizations or people could have identical names. For this exercise, we assume that duplicate names are not a problem, or that we can make them unique by adding a middle initial or a title.

Telephone numbers are also shown as user-assigned primary keys. Addresses, however, are given system-assigned primary keys, because to make an address unique, the primary key must include all the parts such as city, street, and house number. That makes an exceedingly cumbersome key. Addresses are composite keys as well, since parts of an address need to be treated as separate *attributes* (columns) later in the design.

Step 2: Define the Relationships

The model now records the entities, but not the relationships between them. Relationships are not always obvious, but all of the ones worth recording must be found. The only way to ensure that all the relationships are found is to exhaustively list all possible relationships. You must consider every pair of entities *A* and *B* and ask "What is the relationship between an *A* and a *B?*"

Discover the Relationships

The most compact way to discover the relationships is to prepare a matrix that names all the entities on the rows and again on the columns. Figure 8-5 is a matrix that reflects the entities for the personal address book:

	name	address	number (voice)	number (fax)	number (modem)
name					
address					
number (voice)					
number (fax)					
number (modem)					

Figure 8-5   ***A matrix that reflects the entities for a personal address book***

You can ignore the lower triangle of the matrix, as indicated by the shaded area. You must consider the diagonal cells; that is, you must ask the question "What is the relationship between an *A* and another *A?*" In this model, the answer is always none. There is no relationship between a name and a name, or an address and another address, at least none worth recording in this model.

For all cells for which the answer is clearly none, write `none` in the matrix. Now the matrix looks like Figure 8-6.

	name	address	number (voice)	number (fax)	number (modem)
name	none				
address		none			
number (voice)			none		
number (fax)				none	none
number (modem)					none

Figure 8-6 **A matrix in which no entities relate to themselves**

Although in this model no entities relate to themselves, in other models this is not always true. A typical example is the employee who is the manager of another employee. Another example occurs in manufacturing, when a part entity is a component of another part.

Another decision reflected in the matrix is that there is no relationship between a fax number and a modem number.

In the remaining cells, you write the kind of relationship that exists between the entity on the row and the entity on the column. Three kinds of relationships are possible:

- *One-to-one* (written 1:1), in which there is never more than one entity *A* for one entity *B*, and never more than one *B* for one *A*.

- *One-to-many* (written 1:n), in which there is never more than one entity *A*, but there can be several entities *B* related to it (or vice versa).

- *Many-to-many* (written m:n), in which several entities *A* can be related to one *B* and several entities *B* can be related to one *A*.

One-to-many relationships are the most common, but examples of all three relationships are in the address-book model.

As you can see in Figure 8-6, the first unfilled cell represents the relationship between names and addresses. What type is this relationship? A name (you decide) can have zero or one address, but no more than one. You write `0-1` opposite **name** and below **address**, as shown in the following diagram:

	name	address	
name	none	**0-1**	

An address, however, can be associated with more than one name. For example, you can know several people at one company, or two or more people who live at the same address.

Can an address be associated with *zero* names? That is, should it be possible for an address to exist when no names use it? You decide that yes, it can. Below **address** and opposite **name**, you write `0-n`, as shown in the following diagram:

	name	***address***	
name	none	**0-n** 0-1	

If you decided that an address could not exist without being associated with at least one name, you write `1-n` instead of `0-n`.

When a relationship is limited on either side to `1`, it is a 1:n relationship. The relationship between names and addresses is a 1:n relationship. Since zero is a possibility, you must allow nulls when the relationship is encoded as a column in a table.

Now consider the next cell, the relationship between a name and a voice number. How many voice numbers can a name be associated with, one or more than one? Glancing at your address book, you see that often you have noted more than one telephone number for a person—for some busy sales-man you know a home number, an office number, a paging number, and a car phone. But there also can be names with zero associated numbers. You write `0-n` opposite **name** and below **number (voice)**, as shown in the following diagram:

	name	address	number (voice)	
name	none	0-n 0-1	***0-n***	

What of the other side of this relationship? How many names can be associ-ated with a voice number? More than one name, certainly; your address book shows several people at one company for which there is a single number for incoming calls. Can a number be associated with zero names? No, you decide; there is no point recording a number unless it is used by somebody you know. You write `1-m` under **number (voice)** and opposite **name**. (You use *m* to show that this represents a different greater-than-1 quantity than the *n* already noted in that cell.)

	name	address	***number (voice)***	
name	none	0-n 0-1	***1-m*** 0-n	

This is a many-to-many or m:n relationship. On one side, it does not allow zero, which means that when it is translated to a column definition, no nulls are allowed.

Fill out the rest of the matrix in the same fashion, as shown in Figure 8-7.

	name	address	number (voice)	number (fax)	number (modem)
name	none	0-n 0-1	1-m 0-n	1-m 0-n	1 0-n
address		none	none	0-1 0-n	0-1 0-n
number (voice)			none	0-1 0-1	none
number (fax)				none	none
number (modem)					none

Figure 8-7   A completed matrix for an address book

The following decisions are reflected in Figure 8-7:

- A name can be associated with more than one fax number; for example, a company can have several fax machines. Going the other way, a fax number can be associated with more than one name; for example, several people can use the same fax number.

- A modem number must be associated with exactly one name. (This is an arbitrary decree to complicate the example; pretend it is a requirement of the design.) However, a name can have more than one associated modem number; for example, a company computer can have several dial-up lines.

- While there is some relationship between a voice number and an address in the real world, none needs to be recorded in this model. There already is an indirect relationship through *name*.

- There can be more than one modem or fax at a given address; on the other hand, a given modem or fax has one address and there can be modems or faxes whose addresses are not known.

- A voice phone can be associated with one fax phone, and vice versa. This relationship reflects the fact that there is often a voice line to the operator of one of these machines.

You might disagree with some of these decisions (for example, why a relationship between voice numbers and modem numbers is not supported). If you do, revise the matrix and the tables that follow.

Add Relationships to Tables

After you discover all the relationships, you record them in the form of new columns and tables.

You record a 1:n relationship by inserting a new column in one of the tables that represent entities. Add the column to the table for the entity that associates with just one other entity. The new column contains the foreign key of the other entity. For example, consider the 1:n relationship between names and modem numbers in the following diagram:

A name can associate with *n* modems, but a modem number associates with just one name. Therefore, the new column is added to the **modem** table and it contains a foreign key to the **name** table, as shown in the following diagram:

The flag NN (for never null) documents the decision that a modem must be associated with a name. If you permitted a modem to have no associated name, the NN flag is omitted and null values are permitted in this column of the table.

A 1:1 relationship, such as the relationship between voice and fax numbers, is handled like a 1:n relationship by adding a foreign-key column in the table of one entity to link it to the other entity. With a 1:1 relationship, you can put the foreign key in either table. To minimize the space used by the tables, put it in the table you expect to have fewer rows.

You cannot record an m:n relationship by adding columns to entity tables. This relationship requires a table of its own. For example, you decided that a name can have multiple fax numbers and a fax number can serve multiple names. You must add such a relationship to the model as a two-column table. Each column is a foreign key to one of the entities involved.

Name

NameString	
PK UA	
Gaius C. Caesar SPQR, S.A.	

Fax

FaxNumber	
PK UA	
926-6741 61-62-435-143	

NameFax

NameString	FaxNumber	
PK FK NN	PK FK NN	
Gaius C. Caesar SPQR, S.A.	926-6741 61-62-435-143	
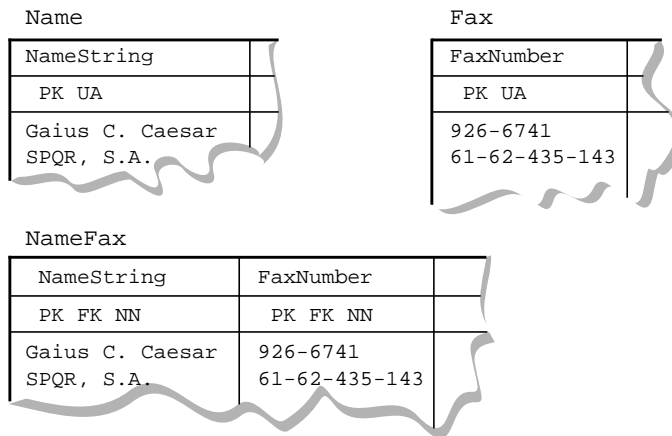
The primary key comprises both columns. Since the columns are part of the primary key, nulls are not allowed. The NN flag is specified anyway to emphasize the point.

The entire model so far is shown in Figure 8-1. Look for all the relationships from the matrix.

Name

NameString	AddressCode	
PK	FK	
Gaius C. Caesar	1001	
SPQR, S.A.	1020	

Address

AddressCode
PK SA
1001
1020

Voice

VceNumber
PK UA
1-800-274-8184
011-49-89-922-030

Fax

FaxNumber	AddressCode	VceNumber
PK UA	FK	FK
926-6741	1001	926-6300
61-62-435-143	1020	(null)

NameFax

NameString	FaxNumber
PK FK	PK FK
Gaius C. Caesar	926-6741
SPQR, S.A.	61-62-435-143

NameVce

NameString	VceNumber
PK FK	PK FK
Gaius C. Caesar	1-800-274-8184
SPQR, S.A.	011-49-89-922-030

Modem

MdmNumber	NameString	VceNumber	AddressCode
PK UA	FK NN	FK	FK
416-566-7024	Nite Owl BBS	926-6300	1001
33 1 42 70 67 74	CompuServe	(null)	1020

Figure 8-8   *The data model with relationships between entities defined*

Step 3: List the Attributes

When all entities and relationships are found, the model contains all the tables it can have. However, entities have *attributes*, that is, characteristics, qualities, amounts, or features. These attributes are added to the model as new columns.

Select Attributes

In selecting attributes, choose ones that have the following qualities:

- They are significant.

  Include only attributes that are important to the users of the database.

- They are direct, not derived.

  An attribute that can be derived from existing attributes—for instance, through an expression in a SELECT statement—should not be made part of the model. The presence of derived data greatly complicates the maintenance of a database.

  At a later stage of the design (discussed in Chapter 10), you can consider adding derived attributes to improve performance, but at this stage you should exclude them.

Select Attribute Tables

You also must be sure to place each attribute in the correct table. An attribute can be part of a table only if its value depends solely on the values of the primary- and candidate-key columns of that table.

To say that the value of an attribute *depends on* a column means that, if the value in the column changes, the value of the attribute must also change. The attribute is a function of the column. The following explanations make this more specific:

- If the table has a one-column primary key, the attribute must depend on that key.
- If the table has a composite primary key, the attribute must depend on the values in all of its columns taken as a whole, not on just one or some of them.
- If the attribute also depends on other columns, these must be columns of a candidate key, that is, columns that are unique in every row.

When you follow these rules, the tables of the model are in what E.F. Codd, the inventor of relational databases, calls *third-normal form*. When tables are not in third-normal form, there is either redundant data in the model or there are problems when you attempt to update the tables.

If you cannot find a place for an attribute that observes these rules, then most likely you have made one of these errors:

- The attribute is not well-defined.
- The attribute is derived, not direct.
- The attribute is really an entity or a relationship.
- Some entity or relationship is missing from the model.

The following attributes were added to the address-book model to produce the tables shown in Figure 8-1 at the start of this chapter:

- Street, city, state or province, and postal code were added to the *address* entity.
- Birth date was added to the *name* entity.
- Type was added to the *voice* entity to distinguish car phones, home phones, and office phones.
- The hours a fax machine is attended was added to the *fax* entity.
- Whether a modem supports 300, 1200, or 2400 baud rates was added to the *modem* entity.

Summary

This chapter summarized and illustrated the following three principle steps of data modeling, as prescribed by Extended Relational Analysis:

1. List the entities that are to be recorded, making sure that they are
   - Significant
   - Generic
   - Fundamental
   - Unitary

   Then, identify their primary keys and document each entity as a table.

2. Exhaustively list the relationship between each pair of entities, classifying it as

    •   None, no relationship

    •   One-to-one

    •   One-to-many

    •   Many-to-many

    Then, record each relationship by adding columns to the entity tables or, for many-to-many relationships, by adding new, two-column tables.

3. Record important attributes of entities or relationships by adding new columns to the tables.

When the process is done right, you are forced to examine every aspect of the data not once, but several times.

Implementing the Model

9

Chapter Overview

Once a data model is prepared, it must be implemented as a database and tables. This chapter covers the decisions that you must make to implement the model.

The first step in implementation is to *complete* the data model by defining a *domain*, or set of data values, for every column. The second step is to implement the model using SQL statements.

The first section of the chapter covers defining domains in detail. The second section shows how you create the database (using the CREATE DATABASE and CREATE TABLE commands) and populate it with data.

Defining the Domains

To complete the data model described in Chapter 8 of this book, you must define a domain for each column. The *domain of a column* is the set of all data values that can properly appear in that column.

The purpose of a domain is to guard the *semantic integrity* of the data in the model, that is, to ensure that it reflects reality in a sensible way. If a name can be entered where a telephone number was planned, or a fractional number where an integer should be, the integrity of the data model is at risk.

You define a domain by first defining the *constraints* that a data value must satisfy before it can be part of the domain. Column domains are specified using the following constraints:

- Data types
- Default values
- Check constraints

In addition, referential constraints can be placed on columns by identifying the primary and foreign keys in each table. How to identify these keys was discussed in Chapter 8 of this manual.

Data Types

The first constraint on any column is the one that is implicit in the data type for the column. When you choose a data type, you constrain the column so that it contains only values that can be represented by that type.

Each data type represents certain kinds of information and not others. The correct data type for a column is the one that represents all the data values that are proper for that column, but as few as possible of the values that are not proper for it.

Choosing a Data Type

Every column in a table must have a data type chosen from the types that the database server supports. The choice of data type is important for several reasons:

- It establishes the basic domain of the column, that is, the set of valid data items the column can store.
- It determines the kinds of operations you can perform on the data. For example, you cannot apply aggregate functions, such as SUM, to columns with a character data type.
- It determines how much space each data item occupies on disk. This is not important for small tables, but if a table has tens of thousands or hundreds of thousands of rows, the difference between a 4-byte and an 8-byte type can be crucial.

The decision tree shown in Figure 9-1 summarizes the choices among data types. They are explained in the following sections.
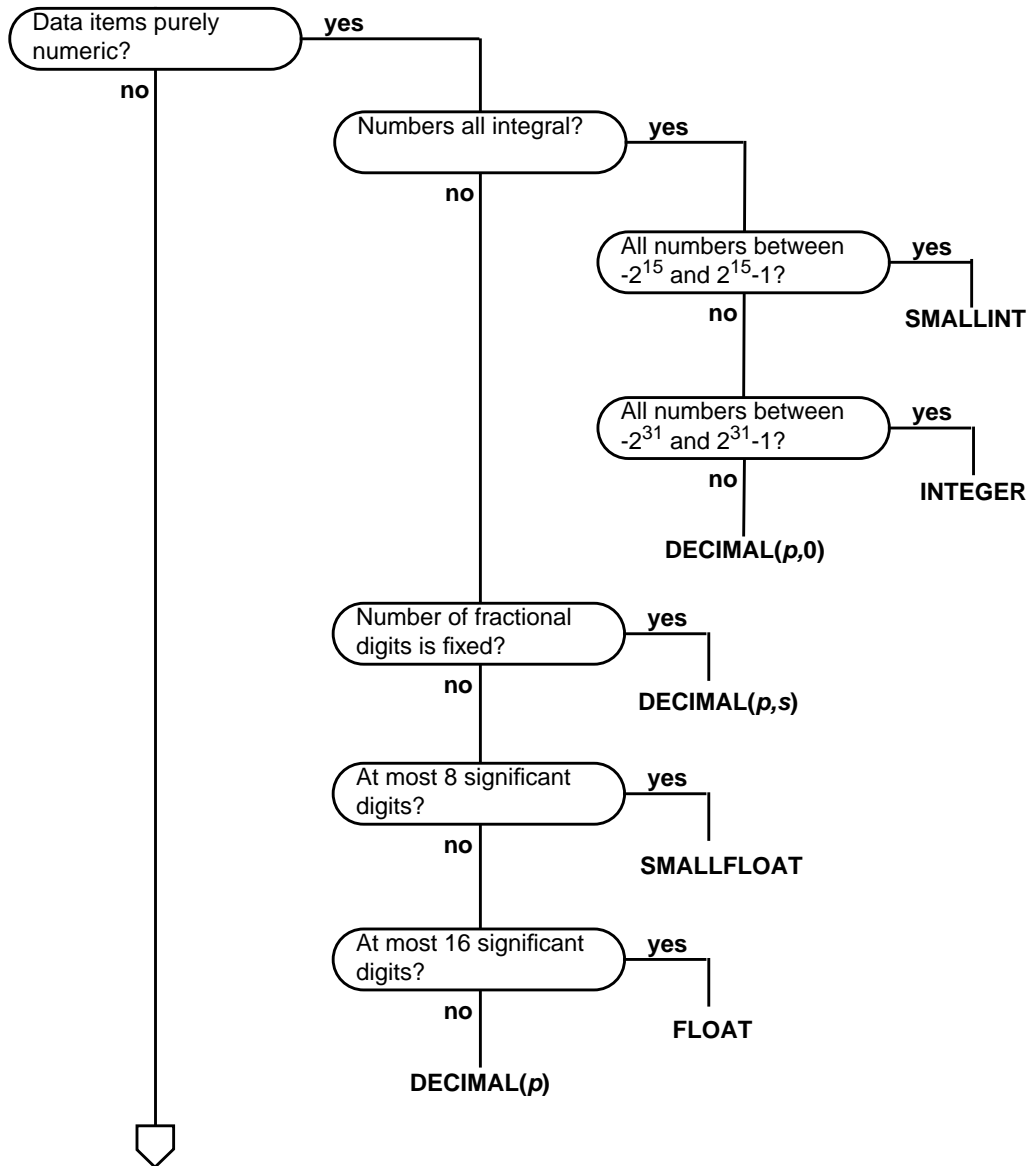
Figure 9-1   *A diagram of the decisions to be made in choosing a data type (1 of 2)*
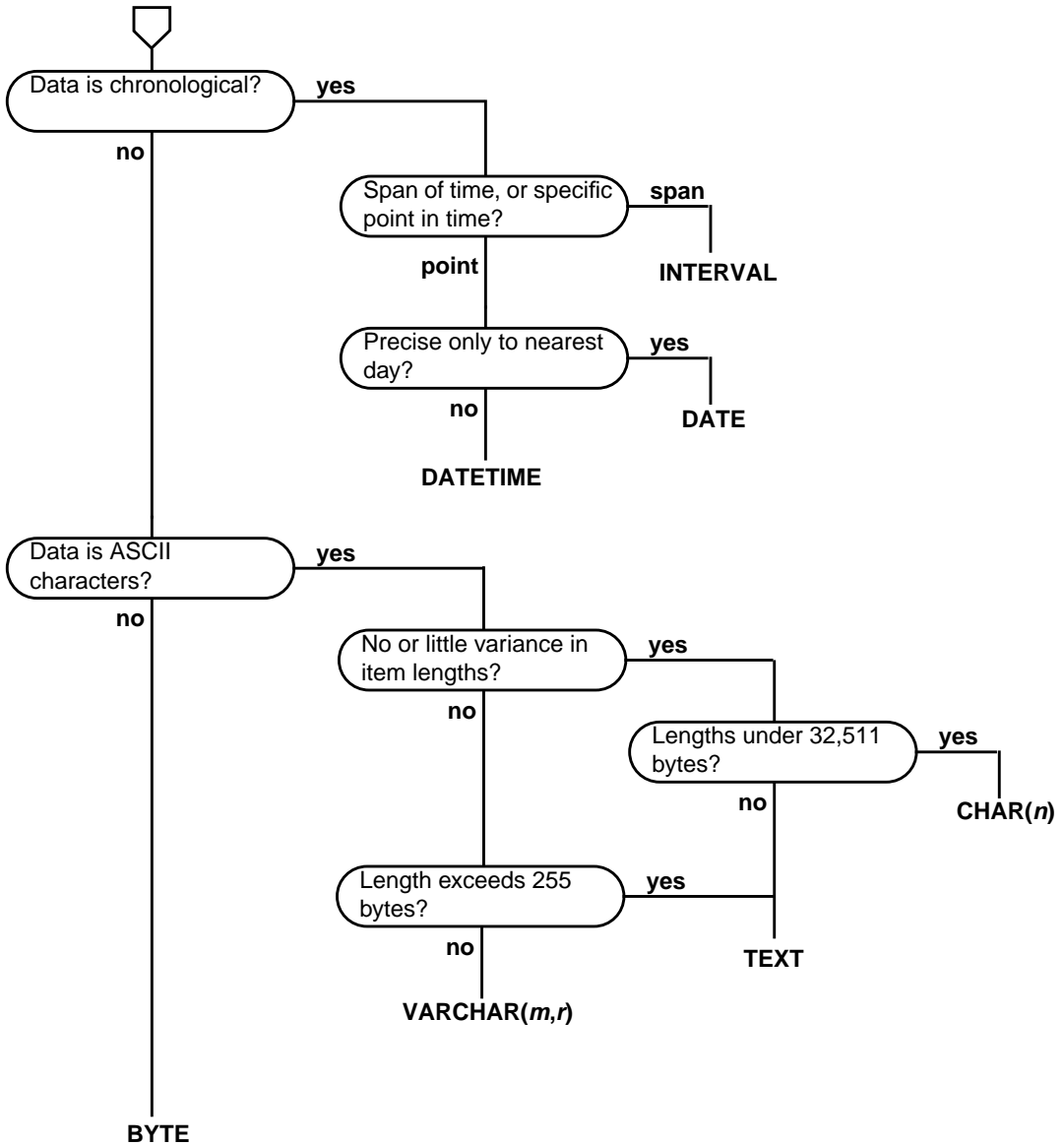
Figure 9-1   *A diagram of the decisions to be made in choosing a data type (2 of 2)*

Numeric Types

Informix database servers support eight numeric data types. Some are best suited for counters and codes, some for engineering quantities, and some for money.

Counters and Codes: INTEGER and SMALLINT

The INTEGER and SMALLINT data types hold small whole numbers. They are suited for columns that contain counts, sequence numbers, numeric identity codes, or any range of whole numbers when you know in advance the maximum and minimum values to be stored.

Both types are stored as signed binary integers. INTEGER values have 32 bits and can represent whole numbers from -2^{31} through $2^{31}-1$; that is, from –2,147,483,647 through 2,147,483,647. (The maximum negative number, –2,147,483,248, is reserved and cannot be used.)

SMALLINT values have only 16 bits. They can represent whole numbers from –32,767 through 32,767. (The maximum negative number, -32,768, is reserved and cannot be used.)

These data types have two advantages:

- They take up little space (2 bytes per value for SMALLINT and 4 bytes per value for INTEGER).
- Arithmetic expressions like SUM and MAX, and sort comparisons, can be done very efficiently on them.

The disadvantage to using INTEGER and SMALLINT is the limited range of values they can store. The database server does not store a value that exceeds the capacity of an integer. Of course, this is not a problem when you know the maximum and minimum values to be stored.

Automatic Sequences: SERIAL

The SERIAL data type is simply INTEGER with a special feature. Whenever a new row is inserted into a table, the database server automatically generates a new value for a SERIAL column. A table can have only one SERIAL column. Because the database server generates them, the serial values in new rows are

always different even when multiple users are adding rows at the same time. This is a useful service, since it is quite difficult for an ordinary program to coin unique numeric codes under those conditions.

How Many Serialized Rows?

After inserting 2^{31} rows in a table, the database server uses up all of the positive serial numbers. Should you be concerned about this? Probably not, since to make this happen you need to insert a row every second for 68 years. However, if it did occur, the database server would continue generating new numbers. It would treat the next-serial quantity as a signed integer. Since it only uses positive values, it would simply wrap around and start generating integer values beginning with 1.

The sequence of generated numbers always increases. When rows are deleted from the table their serial numbers are not reused. This means that rows sorted on a SERIAL column are returned in the order in which they were created. That cannot be said of any other data type.

You can specify the initial value in a SERIAL column in the CREATE TABLE statement. This makes it possible to generate different subsequences of system-assigned keys in different tables. The **stores5** database uses this technique. In **stores5**, the customer numbers begin at 101, while the order numbers start at 1001. As long as this small business does not register more than 899 customers, all customer numbers have three digits and order numbers have four.

A SERIAL column is not automatically a unique column. If you want to be perfectly sure there are no duplicate serial numbers, you must apply a unique constraint (see "Using CREATE TABLE" on page 9-26). However, if you define the table using the interactive schema editor of **DB-Access** or **INFOR-MIX-SQL**, it automatically applies a unique constraint to any SERIAL column.

The SERIAL data type has the following advantages:

- It provides a convenient way to generate system-assigned keys.
- It produces unique numeric codes even when multiple users are updating the table.
- Different tables can use different ranges of numbers.

It has the following disadvantages:

- Only one SERIAL column is permitted in a table.
- It can produce only arbitrary numbers (then again, arbitrary numeric codes might not be acceptable to the database users).

Altering the Next SERIAL Number

The starting value for a SERIAL column is set when the column is created (see "Using CREATE TABLE" on page 9-26). You can use the ALTER TABLE command later to reset the *next* value, the value that is used for the next-inserted row.

You cannot set the *next* value below the current maximum value in the column because that could lead the database server to generate duplicate numbers. However, you can set the *next* value to any value higher than the current maximum, thus creating gaps in the sequence.

Approximate Numbers: FLOAT and SMALLFLOAT

In scientific, engineering, and statistical applications, numbers are often known to only a few digits of accuracy and the magnitude of a number is as important as its exact digits.

The floating-point data types are designed for these applications. They can represent any numerical quantity, fractional or whole, over a wide range of magnitudes from the cosmic to the microscopic. For example, they can easily represent both the average distance from the Earth to the Sun (1.5×10^9 meters) or Planck's constant (6.625×10^{-27}). Their only restriction is their limited precision. Floating-point numbers retain only the most significant digits of their value. If a value has no more digits than a floating-point number can store, the value is stored exactly. If it has more digits, it is stored in approximate form, with its least-significant digits treated as zeros.

This lack of exactitude is fine for many uses, but you should never use a floating-point data type to record money, or any other quantity for which it is an error to change the least significant digits to zero.

There are two sizes of floating-point data types. The FLOAT type is a double-precision, binary floating-point number as implemented in the C language on your computer. One usually takes up 8 bytes. The SMALLFLOAT (also known as REAL) type is a single-precision, binary floating-point number that usually takes up 4 bytes. The main difference between the two data types is their precision. A FLOAT column retains about 16 digits of its values, while a SMALLFLOAT column retains only about 8 digits.

Floating-point numbers have the following advantages:

- They store very large and very small numbers, including fractional ones.
- They represent numbers compactly in 4 or 8 bytes.
- Arithmetic functions such as AVG and MIN, and sort comparisons, are efficient on these data types.

The main disadvantage of floating point numbers is that digits outside their range of precision are treated as zeros.

Adjustable-Precision Floating Point: DECIMAL(*p*)

The DECIMAL(*p*) data type is a floating-point type similar to FLOAT and SMALLFLOAT. The important difference is that you specify how many significant digits it retains. The precision you write as *p* may range from 1 to 32, from fewer than SMALLFLOAT up to twice the precision of FLOAT.

The magnitude of a DECIMAL(*p*) number ranges from 10^{-128} to 10^{126}.

It is easy to be confused about decimal data types. The one under discussion is DECIMAL(*p*); that is, DECIMAL with only a precision specified. The size of DECIMAL(*p*) numbers depends on their precision; they occupy $1+p/2$ bytes (rounded up to a whole number, if necessary).

DECIMAL(*p*) has the following advantages over FLOAT:

- Precision can be set to suit the application, from highly approximate to highly precise.
- Numbers with as many as 32 digits can be represented exactly.
- Storage is used in proportion to the precision of the number.
- Every Informix database server supports the same precision and range of magnitudes regardless of the host operating system.

The DECIMAL(*p*) data type has the following disadvantages:

- Arithmetic and sorting are somewhat slower than on FLOAT numbers.
- Many programming languages do not support the DECIMAL(*p*) data format the way they support FLOAT and INTEGER. When a program extracts a DECIMAL(*p*) value from the database, it may have to convert the value to another format for processing. (However, **INFORMIX-4GL** programs can use DECIMAL(*p*) values directly.)

Fixed-Point Numbers: DECIMAL and MONEY

Most commercial applications need to store numbers that have fixed numbers of digits on the right and left of the decimal point. Amounts of money are the most common examples. Amounts in U.S. and other currencies are written with two digits to the right of the decimal point. Normally, you also know the number of digits needed on the left, depending on whose transactions are recorded—perhaps 5 digits for a personal budget, 7 for a small business, and 12 or 13 for a national budget.

These numbers are *fixed-point* numbers because the decimal point is fixed at a specific place regardless of the value of the number. The DECIMAL(*p,s*) data type is designed to hold them. When you specify a column of this type, you write its *precision* (*p*) as the total number of digits it can store, from 1 to 32. You write its *scale* (*s*) as the number of those digits that fall to the right of the decimal point. (The relation between precision and scale is diagrammed in Figure 9-2.) Scale can be zero, meaning it stores only whole numbers. When this is done, DECIMAL(*p,s*) provides a way of storing integers of up to 32 digits.
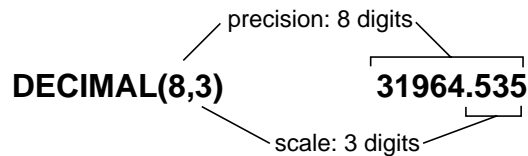
Figure 9-2   ***The relation between precision and scale in a fixed-point number***

Like the DECIMAL(*p*) data type, DECIMAL(*p,s*) takes up space in proportion to its precision. One value occupies $1+p/2$ bytes, rounded up to a whole number of bytes.

The MONEY type is identical to DECIMAL(*p,s*), but with one extra feature. Whenever the database server converts a MONEY value to characters for display, it automatically includes a currency symbol.

The advantages of DECIMAL(*p,s*) over INTEGER and FLOAT are that much greater precision is available (up to 32 digits as compared to 10 for INTEGER and 16 for FLOAT), while both the precision and the amount of storage required can be adjusted to suit the application.

The disadvantages are that arithmetic is less efficient and that many pro-gramming languages do not support numbers in this form. Therefore, when a program extracts a number, it usually must convert it to another numeric form for processing. (However, **INFORMIX-4GL** programs can use DECIMAL(p,s) and MONEY values directly.)

Whose Money? Choosing a Currency Format

Each nation has its own way of punctuating amounts of money. When an Informix database server displays a MONEY value, it refers to a currency format defined to the operating system, usually in a variable named DBMONEY. A currency symbol can precede or follow the amount, and the decimal delimiter can be set to a period, a comma, or another character. For details, see Chapter 4 of *The Informix Guide to SQL: Reference.*

Chronological Types

Informix database servers support three data types for recording time. The DATE data type stores a calendar date. DATETIME records a point in time to any degree of precision from a year to a fraction of a second. The INTERVAL data type stores a span of time; that is, a duration.

Calendar Dates: DATE

The DATE data type stores a calendar date. A DATE value is actually a signed integer whose contents are interpreted as a count of full days since midnight on December 31, 1899. Most often it holds a positive count of days into the current century.

The DATE format has ample precision to carry dates into the far future (58,000 centuries). Negative DATE values are interpreted as counts of days prior to the epoch date; that is, a DATE of -1 represents the day December 30, 1899.

Since DATE values are integers, Informix database servers permit them to be used in arithmetic expressions. For example, you can take the average of a DATE column, or you can add 7 or 365 to a DATE column. In addition, there is a rich set of functions specifically for manipulating DATE values. (See Chapter 7 of *The Informix Guide to SQL: Reference.*)

The DATE data type is compact at 4 bytes per item. Arithmetic functions and comparisons execute quickly on a DATE column.

M/D/Y? D-M-Y? Choosing a Date Format

There are many ways to punctuate and order the components of a date. When an Informix database server displays a DATE value, it refers to a format defined to the operating system, usually in a variable named DBDATE. The numbers for day, month, and year can be shown in any order, separated by a chosen delimiter. For more information, see Chapter 4 of *The Informix Guide to SQL: Reference.*

Exact Points in Time: DATETIME

The DATETIME data type stores any moment in time in the era beginning 1 A.D. In fact, DATETIME is really a family of 28 data types, each with a different precision. When you define a DATETIME column, you specify its precision. It may contain any sequence from the list *year, month, day, hour, minute, second,* and *fraction.* Thus, you can define a DATETIME column that stores only a year, or only a month and day, or a date and time that is exact to the hour or even to the millisecond. The size of a DATETIME value ranges from 2 to 11 bytes depending on its precision, as shown in Figure 9-3.

The advantage of DATETIME is that it can store dates more precisely than to the nearest day, and it can store time values. Its only disadvantage is an inflexible display format, but this can be circumvented (see "Forcing the Format of a DATETIME or INTERVAL Value" on page 9-15).

Precision	Size*	Precision	Size*
year to year	3	day to hour	3
year to month	4	day to minute	4
year to day	5	day to second	5
year to hour	6	day to fraction(f)	$5+f/2$
year to minute	7	hour to hour	2
year to second	8	hour to minute	3
year to fraction (f)	$8+f/2$	hour to second	4
month to month	2	hour to fraction(f)	$4+f/2$
month to day	3	minute to minute	2
month to hour	4	minute to second	3
month to minute	5	minute to fraction(f)	$3+f/2$
month to second	6	second to second	2
month to fraction(f)	$6+f/2$	second to fraction(f)	$2+f/2$
day to day	2	fraction to fraction(f)	$1+f/2$

* When *f* is odd, round the size to the next full byte.

Figure 9-3   *All possible precisions of the DATETIME data type, with their sizes in bytes*

Durations: INTERVAL

The INTERVAL data type stores a duration, that is, a length of time. The difference between two DATETIME values is an INTERVAL that represents the span of time that separates them. Here are some examples that might help to clarify the differences:

- An employee began working here on May 21, 1991 (either a DATE or a DATETIME).

- She worked here 254 days (an INTERVAL, the difference between the TODAY function and the starting DATE or DATETIME).

- She begins work each day at 0900 hours (a DATETIME).

- She works 8 hours (an INTERVAL) with 45 minutes for lunch (another INTERVAL).

- Her quitting time is 1745 hours (the sum of the DATETIME when she begins work and the two INTERVALs).

Like DATETIME, INTERVAL is a family of types with different precisions. An INTERVAL can represent a count of years and months; or it can represent a count of days, hours, minutes, seconds, or fractions of seconds: 18 possible precisions in all. The size of an INTERVAL value ranges from 2 to 8 bytes, as shown in Figure 9-4.

Precision	Size*	Precision	Size*
year(p) to year	$1+p/2$	hour(p) to minute	$2+p/2$
year(p) to month	$2+p/2$	hour(p) to second	$3+p/2$
month(p) to month	$1+p/2$	hour(p) to fraction(f)	$3+(p+f)/2$
day(p) to day	$1+p/2$	minute(p) to minute	$1+p/2$
day(p) to hour	$2+p/2$	minute(p) to second	$2+p/2$
day(p) to minute	$3+p/2$	minute(p) to fraction(f)	$2+(p+f)/2$
day(p) to second	$4+p/2$	second(p) to second	$1+p/2$
day(p) to fraction(f)	$4+(p+f)/2$	second(p) to fraction(f)	$1+(p+f)/2$
hour(p) to hour	$1+p/2$	fraction to fraction(f)	$1+f/2$

*Round a fractional size to the next full byte.

Figure 9-4   ***All possible precisions of the INTERVAL data type, with their sizes in bytes***

INTERVAL values can be negative as well as positive. You can add or subtract them, and you can scale them by multiplying or dividing by a number. This is not true of either DATE or DATETIME. It is reasonable to ask, "What is one-half the number of days until April 23?" but it is not reasonable to ask, "What is one-half of April 23?"

Character Types

All Informix database servers support the CHAR(*n*) data type. **INFOR-MIX-OnLine** supports three other data types that have special uses.

Character Data: CHAR(*n*)

The CHAR(*n*) data type contains a sequence of *n* ASCII characters. The length *n* ranges from 1 to 32,511.

Forcing the Format of a DATETIME or INTERVAL Value

The database server always displays the components of an INTERVAL or DATETIME value in the order *year-month-day hour:minute:second.fraction.* It does not refer to the date format defined to the operating system, as it does when formatting a DATE.

You can write a SELECT statement that displays the date part of a DATETIME value in the system-defined format. The trick is to isolate the component fields using the EXTEND function and pass them through the MDY() function, which converts them to a DATE. Here is a partial example:

```
SELECT ... MDY (
     EXTEND (DATE_RECEIVED, MONTH TO
      MONTH),
     EXTEND (DATE_RECEIVED, DAY TO DAY),
     EXTEND (DATE_RECEIVED, YEAR TO YEAR))
     FROM RECEIPTS ...
```

When you are designing a report using **INFORMIX-4GL** or **INFOR-MIX-SQL**, you have the greater flexibility of the PRINT statement. Select each component of a DATETIME or INTERVAL value as an expression using EXTEND. Give each expression an alias for convenience. Combine the components in a PRINT expression with the desired punctuation.

```
SELECT ...
     EXTEND (START_TIME, HOUR TO HOUR) H,
     EXTEND (START_TIME, MINUTE TO MINUTE) M, ...
```

Then, in the report:

```
PRINT "Start work at ", H USING "&&", M USING "&&", "hours."
```

Whenever a CHAR(*n*) value is retrieved or stored, exactly *n* bytes are transferred. If an inserted value is short, it is extended with spaces to make up *n* bytes. A value longer than *n* is truncated. There is no provision for variable-length data in this format.

A CHAR(*n*) value can include tabs and spaces but normally contains no other nonprinting characters. When rows are inserted using INSERT or UPDATE, or when rows are loaded with a utility program, there is no means of entering nonprintable characters. However, when rows are created by a program using embedded SQL, the program can insert any character except the null (binary zero) character. It is not a good idea to store nonprintable characters in a character column since standard programs and utilities do not expect them.

The advantage of the CHAR(*n*) data type is that it is available on all database servers. Its only disadvantage is that its length is fixed. When the length of data values varies widely from row to row, space is wasted.

Variable-Length Strings: VARCHAR(*m,r*)

Often the items in a character column have differing lengths; that is, many have an average length and only a few have the maximum length. The VARCHAR(*m,r*) data type is designed to save disk space when storing such data. A column defined as VARCHAR(*m,r*) is used just like one defined as CHAR(*n*). When you define a VARCHAR(*m,r*) column, you specify *m* as the *maximum* length of a data item. Only the actual contents of each item are stored on disk, with a one-byte length field. The limit on *m* is 255; that is, a VARCHAR(*m,r*) value can store up to 255 characters. If the column is indexed, the limit on *m* is 254.

The second parameter, *r*, is an optional *reserve* length that sets a lower limit on the length of an item as stored on disk. When an item is shorter than *r*, *r* bytes are nevertheless allocated to hold it. The purpose is to save time when rows are updated. (See "Variable-Length Execution Time, Too" on page 9-17.)

The advantages of the VARCHAR(*m,r*) data type over the CHAR(*n*) type are as follows:

- It conserves disk space when the lengths of data items vary widely, or when only a few items are longer than average.
- Queries on the more compact tables can be faster.

Its disadvantages are as follows:

- It does not allow lengths that exceed 255 characters.
- Updates of a table can be slower in some circumstances.
- It is not available with all Informix database servers.

Variable-Length Execution Time, Too

When the VARCHAR(m,r) data type is used, the rows of a table have varying lengths instead of fixed lengths. This has mixed effects on the speed of database operations.

Since more rows fit in a disk page, the database server can search the table using fewer disk operations than if the rows were of fixed length. As a result, queries can execute more quickly. Insert and delete operations can be a little quicker for the same reason.

When you update a row, the amount of work the database server must do depends on the length of the new row as compared to the old one. If the new row is the same size or shorter, execution time is not significantly different than it is with fixed-length rows. But if the new row is longer than the old one, the database server might have to perform several times as many disk operations. Thus, updates of a table that uses VARCHAR(m,r) data can sometimes be slower than updates of a fixed-length field.

You can mitigate this effect by specifying *r* as a size that covers a high proportion of the data items. Then, most rows use the reserve length; only a little space is wasted in padding, and updates are only slow when a normal value is replaced with one of the rare, longer ones.

Large Character Objects: TEXT

The TEXT data type stores a block of text. It is designed to store self-contained documents: business forms, program source or data files, or memos. Although you can store any data at all in a TEXT item, Informix tools expect that a TEXT item is printable, so this data type should be restricted to printable ASCII text.

TEXT values are not stored with the rows of which they are a part. They are allocated in whole disk pages, usually areas away from rows. (See "Locating Blob Data" on page 10-18.)

How Blobs Are Used

Collectively, columns of type TEXT and BYTE are called *Binary Large Objects*, or blobs. The database server only stores and retrieves them. Normally, blob values are fetched and stored by programs written using **INFORMIX-4GL** or a language that supports embedded SQL, such as **INFORMIX-ESQL/C**. In such a program you can fetch, insert, or update a blob value in a manner similar to the way you read or write a sequential file.

In any SQL statement, interactive or programmed, a blob column *cannot* be used in the following ways:

- In arithmetic or Boolean expressions
- In a GROUP BY or ORDER BY clause
- In a UNIQUE test
- For indexing, either by itself or as part of a composite index

In a SELECT statement entered interactively, or in a form or report, a blob can only

- Be selected by name, optionally with a subscript to extract part of it
- Have its length returned by selecting LENGTH(*column*)
- Be tested with the IS [NOT] NULL predicate

In an interactive INSERT statement, you can use the VALUES clause to insert a blob value, but the only value you can give that column is null. However, you can use the SELECT form of the INSERT statement to copy a blob value from another table.

In an interactive UPDATE statement, you can update a blob column to null or to a subquery returning a blob column.

The advantage of the TEXT data type over CHAR(*n*) and VARCHAR(*m,r*) is that the size of a TEXT data item has no limit except the capacity of disk storage to hold it. The disadvantages of the TEXT data type are as follows:

- It is allocated in whole disk pages; a short item wastes space.
- There are restrictions on how you can use a TEXT column in an SQL statement. (See "How Blobs Are Used" on page 9-18.)
- It is not available with all Informix database servers.

You can display TEXT values in reports generated with **INFORMIX**-**4GL** programs or the **ACE** report writer. You can display TEXT values on a screen and edit them using screen forms generated with **INFORMIX**-**4GL** programs or with the **PERFORM** screen-form processor.

Binary Objects: BYTE

The BYTE data type is designed to hold any data a program can generate: graphic images, program object files, and documents saved by any word processor or spreadsheet. The database server permits any kind of data of any length in a BYTE column.

Like TEXT, BYTE data items are stored in whole disk pages in separate disk areas from normal row data.

The advantage of the BYTE data type, as opposed to TEXT or CHAR(*n*), is that it accepts any data. Its disadvantages are the same as those of the TEXT data type.

Changing the Data Type

You can use the ALTER TABLE command to change the data type assigned to a column after the table is built. While this is sometimes necessary, you should avoid it for the following reasons:

- To change a data type, the database server must copy and rebuild the table. For large tables, that can take a lot of time and disk space.
- Some changes of data type can cause a loss of information. For example, when you change a column from a longer to a shorter character type, long values are truncated; when you change to a less-precise numeric type, low-order digits are truncated.
- Existing programs, forms, reports, and stored queries also might have to be changed.

Default Values

A default value is the value inserted into a column when an explicit value is not specified. A default value can be a literal character string defined by you or one of the following SQL null, constant expressions:

- USER
- CURRENT
- TODAY
- SITENAME

Not all columns need default values, but as you work with your data model you may discover instances where the use of a default value saves data-entry time or prevents data-entry error. For example, the address-book model has a State column. While looking at the data for this column, you discover that more than 50% of the addresses list California as the state. To save time, you specify the string "California" as the default value for the State column.

Default values also come in handy when a column contains a flag. The address-book model uses a flag (Y/N) to indicate whether a modem works at 300, 1200, or 2400 baud. You could specify a default value ("N") for that column. Then you no longer need to enter a value for those modems that do not work at certain baud rates.

Check Constraints

Check constraints specify a condition or requirement on a data value before it can be a part of a domain. You can write a constraint in natural language, like the previous examples. Or for greater precision, you also can write a constraint in the syntax of a WHERE clause. For example, the following requirement constrains the values of an integer domain to a certain range:

```
Customer_Number >= 50000 AND Customer_Number <= 99999
```

You can express constraints on character-based domains this way using the MATCHES predicate and the regular-expression syntax it supports. For example, the following constraint restricts the Telephone domain to the form of a U.S. local telephone number:

```
VceNumber MATCHES "[2-9][2-9][0-9]-[0-9][0-9][0-9][0-9]"
```

However, constraints in this form can be tedious to write and hard to read. Writing the rule in plain language is easier.

Specifying Domains

One domain can apply to many columns. In the address-book model, there are several columns whose domain is telephone numbers. You only need to specify the Telephone domain once.

The final step of creating a data model is to specify the domains needed in the model. The result is a list of domains with the following columns:

- A name, for ease of reference
- The data type; for instance, INTEGER or CHAR(*n*).
- The additional rules that constrain data of this kind, written in either WHERE-clause form or English.

One possible list of domains for the address-book data model is shown in Figure 9-5.

Domain	Data Type	Other Constraints
Address code	SERIAL	1000<code AND code <10000
Attendant hours	DATETIME HOUR TO MINUTE	
Birthday	DATE	birthday <= TODAY
City	VARCHAR(40,10)	
Name	VARCHAR(50,20)	
Postal code	CHAR(10)	only valid postal codes
State/province	CHAR(5)	
Street address	VARCHAR(50,20)	
Telephone	CHAR(13)	only valid phone numbers
True/false	CHAR(1)	Y = true, N = false, DEFAULT = N
Voice phone type	CHAR(10)	

Figure 9-5   *The domains needed to complete the address-book data model*

When you complete the list, write the names of the domains under the columns that use them in the display of the tables of the model. For instance, you write the domain Telephone under six columns in five tables of the address-book model. Figure 9-6 shows the complete model.

NameString	AddressCode	BirthDate
PK UA	FK	
Gaius C. Caesar SPQR, S.A.	1001 1020	17-03-1990 (null)
Name	*Address code*	*Birthday*

Address

AddressCode	Street	City	StaProv	Postcode
PK SA				
1001 1020	2430 Tasso St. #8 411 Bohannon Drive	St. Louis Lenaxa	CA Abta.	83267-1048 TW2 5AQ
Address code	*Street address*	*City*	*State/province*	*Postal code*

Voice

VceNumber	VceType
PK UA	
1-800-274-8184 011-49-89-922-030	home car
Telephone	*Voice phone type*

Fax

FaxNumber	AddressCode	VceNumber	OperFrom	OperTill
PK UA	FK	FK		
926-6741 61-62-435-143	1001 1020	926-6300 (null)	0800 1730	0400 1900
Telephone	*Address code*	*Telephone*	*Attendant hours*	

NameFax

NameString	FaxNumber
PK FK	PK FK
Gaius C. Caesar SPQR, S.A.	926-6741 61-62-435-143
Name	*Telephone*

NameVce

NameString	VceNumber
PK FK	PK FK
Gaius C. Caesar SPQR, S.A.	1-800-274-8184 011-49-89-922-030
Name	*Telephone*

Modem

MdmNumber	NameString	VceNumber	AddressCode	B300	B1200	B2400
PK UA	FK NN	FK	FK			
416-566-7024 33 1 42 70 67 74	Nite Owl BBS CompuServe	926-6300 (null)	1001 1020	Y Y	Y Y	N Y
Telephone	*Name*	*Telephone*	*Address code*	*True/false*		

Figure 9-6   ***The address-book data model begun in Chapter 8, with domains added***

Creating the Database

Now you are ready to create the data model as tables in a database. You do this with the CREATE DATABASE, CREATE TABLE, and CREATE INDEX commands. The syntax of these commands is shown in detail in *The Informix Guide to SQL: Reference.* This section discusses the use of CREATE DATABASE and CREATE TABLE in implementing a data model. The use of CREATE INDEX is covered in the next chapter.

You might have to create the same model more than once. However, the commands that create the model can be stored and executed automatically.

When the tables exist, you must populate them with rows of data. You can do this manually, with a utility program, or with custom programming.

Using CREATE DATABASE

A database is a container that holds all the parts that go into a data model. These parts include the tables, of course, but also the views, indexes, and synonyms. You must create a database before you can create anything else. Besides creating the database, the CREATE DATABASE command establishes the kind of transaction logging to be used. Transaction logging is discussed in Chapter 7.

The **INFORMIX-OnLine** database server differs from other database servers in the way that it creates databases and tables. **INFORMIX-OnLine** is covered first.

Using CREATE DATABASE with INFORMIX-OnLine

When the **INFORMIX-OnLine** database server creates a database, it sets up records that show the existence of the database and its mode of logging. It manages disk space directly, so these records are not visible to operating system commands.

Avoiding Name Conflicts

Normally, only one copy of **INFORMIX-OnLine** is running on a machine, and it manages the databases that belong to all users of that machine. It keeps only one list of database names. The name of your database must be different from that of any other database managed by that database server. (It is possible to run more than one copy of the database server. This is sometimes done, for example, to create a safe environment for testing apart from the operational data. In that case, you must be sure you are using the correct database server when you create the database, and again when you later access it.)

Selecting a Dbspace

INFORMIX-**OnLine** offers you the option of creating the database in a particular *dbspace*. A dbspace is a named area of disk storage. Ask your **INFOR-MIX**-**OnLine** administrator whether you should use a particular dbspace. The administrator can put a database in a dbspace to isolate it from other databases, or to locate it on a particular disk device. (Chapter 10 discusses dbspaces and their relationship to disk devices.)

Some dbspaces are *mirrored* (duplicated on two different disk devices for high reliability); your database can be put in a mirrored dbspace if its contents are of exceptional importance.

Choosing the Type of Logging

INFORMIX-**OnLine** offers the following four choices for transaction logging:

- No logging at all. This choice is not recommended; if the database is lost to a hardware failure, all data alterations since the last archive are lost.

```
CREATE DATABASE db_with_no_log
```

  When you do not choose logging, BEGIN WORK and other SQL statements related to transaction processing are not permitted in the database. This affects the logic of programs that use the database.

- Regular (unbuffered) logging. This is the best choice for most databases. In the event of a failure, only uncommitted transactions are lost.

```
CREATE DATABASE a_logged_db WITH LOG
```

- Buffered logging. If the database is lost, a few of the most recent alterations are lost, or possibly none. In return for this small risk, performance during alterations is slightly improved.

```
CREATE DATABASE buf_log_db WITH BUFFERED LOG
```

  Buffered logging is best for databases that are updated frequently (so that speed of updating is important), but the updates can be re-created from other data in the event of a crash. You can use the SET LOG statement to alternate between buffered and regular logging.

- ANSI-compliant logging. This is the same as regular logging, but the ANSI rules for transaction processing are also enforced. (See the discussion of ANSI SQL in Chapter 1.)

```
CREATE DATABASE std_rules_db WITH LOG MODE ANSI
```

The design of ANSI SQL prohibits the use of buffered logging.

The **INFORMIX-OnLine** administrator can turn transaction logging on and off later. For example, it can be turned off before inserting a large number of new rows.

Using CREATE DATABASE with Other Informix Database Servers

Other Informix database servers create a database as a set of one or more files managed by the operating system. For example, under the UNIX operating system, a database is a small group of files in a directory whose name is the database name. (See the manual for your database server for details on how it uses files.) This means that the rules for database names are the same as the rules the operating system has for file names.

Choosing the Type of Logging

Other database servers offer the following three choices of logging:

- No logging at all. This choice is not recommended; if the database is lost to a hardware failure, all data alterations since the last archive are lost.

```
CREATE DATABASE not_logged_db
```

When you do not choose logging, BEGIN WORK and other SQL statements related to transaction processing are not permitted in the database. This affects the logic of programs that use the database.

- Regular logging. This is the best choice for most databases. If the database is lost, only the alteration in progress at the time of failure is lost.

```
CREATE DATABASE a_logged_db WITH LOG IN "a-log-file"
```

You must specify a file to contain the transaction log. (The form of the file name depends on the rules of your operating system.) This file grows whenever the database is altered. Whenever the database files are

archived, you should set the log file back to an empty condition so it reflects only transactions following the latest archive.

- ANSI-compliant logging. This is the same as regular logging, but the ANSI rules for transaction processing are also enabled. (See the discussion of ANSI-compliant databases in Chapter 1.)

```
CREATE DATABASE std_rules_db WITH LOG IN "a-log-file" MODE
ANSI
```

You can add a transaction log to a nonlogged database later using the START DATABASE statement. Once you apply logging you cannot remove it. (Only **INFORMIX-OnLine** allows logging to be turned on and off.)

Using CREATE TABLE

Use the CREATE TABLE statement to create each table you designed in the data model. This statement has a complicated form, but it is basically a list of the columns of the table. For each column, you supply the following information:

- The name of the column
- The data type (from the domain list you made)
- If the column (or columns) is a primary key, the constraint PRIMARY KEY
- If the column (or columns) is a foreign key, the constraint FOREIGN KEY
- If the column is not a primary key and should not allow nulls, the constraint NOT NULL
- If the column is not a primary key and should not allow duplicates, the constraint UNIQUE
- If the column has a default value, the constraint DEFAULT
- If the column has a check constraint, the constraint CHECK

In short, the CREATE TABLE statement is an image in words of the table as you drew it in the data model diagram. Figure 9-7 shows the statements for the address-book model whose diagram is shown in Figure 9-6 on page 9-22.

```
CREATE TABLE NAME
    (
    NameString      VARCHAR(50,20) PRIMARY KEY,
    AddressCode     INTEGER,
    BirthDate       DATE
    FOREIGN KEY (AddressCode) REFERENCES ADDRESS (AddressCode)
    );
CREATE TABLE ADDRESS
    (
    AddressCode     SERIAL(1000) PRIMARY KEY,
    Street          VARCHAR(50,20),
    City            VARCHAR(40,10),
    StaProv         CHAR(5),
    PostCode        CHAR(10)
    );
CREATE TABLE VOICE
    (
    VceNumber       CHAR(13) PRIMARY KEY,
    VceType         CHAR(10)
    );
CREATE TABLE FAX
    (
    FaxNumber       CHAR(13) PRIMARY KEY,
    AddressCode     INTEGER,
    VceNumber       CHAR(13),
    OperFrom        DATETIME HOUR TO MINUTE,
    OperTill        DATETIME HOUR TO MINUTE
    FOREIGN KEY (AddressCode) REFERENCES ADDRESS (AddressCode),
    FOREIGN KEY (VceNumber) REFERENCES VOICE (VceNumber)
    );
CREATE TABLE NAMEVCE
    (
    NameString      VARCHAR(50,20),
    VceNumber       CHAR(13),
    PRIMARY KEY (NameString, VceNumber),
    FOREIGN KEY (NameString) REFERENCES NAME (NameString),
    FOREIGN KEY (VceNumber) REFERENCES VOICE (VceNumber)
    );
CREATE TABLE NAMEFAX
    (
    NameString      VARCHAR(50,20),
    FaxNumber       CHAR(13),
    PRIMARY KEY (NameString, FaxNumber),
    FOREIGN KEY (NameString) REFERENCES NAME (NameString),
    FOREIGN KEY (FaxNumber) REFERENCES FAX (FaxNumber)
    );
CREATE TABLE MODEM
    (
    MdmNumber       CHAR(13) PRIMARY KEY,
    NameString      VARCHAR(50,20) NOT NULL,
    VceNumber       CHAR(13),
    AddressCode     INTEGER,
    B300            CHAR(1) DEFAULT "N",
    B1200           CHAR(1) DEFAULT "N",
    B2400           CHAR(1) DEFAULT "N",
    FOREIGN KEY (NameString) REFERENCES NAME (NameString),
    FOREIGN KEY (VceNumber) REFERENCES VOICE (VceNumber),
    FOREIGN KEY (AddressCode) REFERENCES ADDRESS (AddressCode)
    );
```

Figure 9-7   The CREATE TABLE statements for the address-book data model

Using Command Scripts

You can create the database and tables by entering the statements interactively. But what if you have to do it again, or several more times?

You might have to do it again to make a production version after a test version is satisfactory. Or you might have to implement the same data model on several machines. To save time and reduce the chance of errors, you can put all the commands to create a database in a file, and execute them automatically.

Capturing the Schema

You can write the statements to implement your model into a file yourself. However, you can also have a program do it for you. See the manual for your database server. It documents the **dbschema** utility, a program that examines the contents of a database and generates all the SQL statements required to re-create it. You can build the first version of your database interactively, making changes until it is exactly as you want it. Then you can use **dbschema** to generate the SQL statements necessary to duplicate it.

Executing the File

DB-**Access** or **INFORMIX**-**SQL**, the programs you use to enter SQL statements interactively, can be driven from a file of commands. The use of these products from the operating system command line is covered in the **DB**-**Access** or **INFORMIX**-**SQL** manuals. You can start **DB**-**Access** or **INFORMIX**-**SQL** and have them read and execute a file of commands prepared by you or by **dbschema**.

An Example

Most Informix products are delivered with a demonstration database called **stores5** (the one used for most of the examples in this book). The **stores5** database is delivered as an operating system command script that calls Informix products to build the database. You can copy this command script and use it as the basis for automating your own data model.

Populating the Tables

For your initial tests, it is easiest to populate the tables interactively by typing INSERT statements to **DB-Access** or **INFORMIX-SQL**. Or, if you are preparing an application program in **INFORMIX-4GL** or another language, you can use the program to enter rows.

Often, the initial rows of a large table can be derived from data held in tables in another database or in operating system files. You can move the data into your new database in a bulk operation. If the data is in another Informix database, you can retrieve it in several ways.

If you are using **INFORMIX-OnLine**, you can simply select the data you want from the other database as part of an INSERT statement in your database. With other database servers, you must export the data to a file. You can use the UNLOAD command of **DB-Access**, **INFORMIX-SQL**, or **INFORMIX-4GL**, or you can write a report in **ACE** or **INFORMIX-4GL** and direct the output to a file.

When the source is another kind of file or database, you must find a way to convert it into a flat ASCII file; that is, a file of printable data in which each line represents the contents of one table row.

Once you have the data in a file, you can load it into a table using the **dbload** utility. Read about **dbload** in the documentation for your database server. The LOAD command of **DB-Access**, **INFORMIX-SQL**, or **INFORMIX-4GL** can also load rows from a flat ASCII file.

Inserting hundreds or thousands of rows goes much faster if you turn off transaction logging. There is no point in logging these insertions anyway, because in the event of a failure you can easily re-create the lost work. Here are the steps of a large bulk-load operation:

- If there is any chance that other users are using the database, exclude them with the DATABASE EXCLUSIVE statement.

- If you are using **INFORMIX-OnLine**, ask the administrator to turn off logging for the database.

  The existing logs can be used to recover the database to its present state, while the bulk insertion can be run again to recover those rows if they are lost soon after.

- Perform the statements or run the utilities that load the tables with data.

- Archive the newly loaded database.

  If you are using **INFORMIX**-**OnLine**, either ask the administrator to per-form a full or incremental archive, or else use the **tbunload** utility to make a binary copy of your database only.

  If you are using other database servers, use operating system commands to back up the files that represent the database.

- Restore transaction logging and release the exclusive lock on the database.

You can enclose the steps of populating a database in a script of operating system commands. You can automate the **INFORMIX**-**OnLine** administrator commands by invoking the command-line equivalents to DB-Monitor.

Summary

This chapter covered the work you must do to implement a data model:

- Specify the domains, or constraints, that are used in the model, and com-plete the model diagram by assigning constraints to each column.
- Use interactive SQL to create the database and the tables in it.
- If you must create the database again, write the SQL statements to do so into a script of commands for the operating system.
- Populate the tables of the model, first using interactive SQL and then by bulk operations.
- Possibly write the bulk-load operation into a command script so you can repeat it easily.

You can now use and test your data model. If it contains very large tables, or if you must protect parts of it from certain classes of users, there is more work to do. That is the subject of the next chapter.

10

Tuning the Model

Chapter Overview

The preceding chapters summarized the steps of creating and implementing a data model that is theoretically sound. Normally, the resulting database also yields adequate performance. If it does, the job is done.

However, some applications have more stringent performance requirements. Some databases contain extremely large tables, others must be usable by many programs concurrently or by programs that operate under tight requirements for response time, and others must be extremely reliable.

You can tune your database design to meet such requirements. In some cases, you might have to sacrifice theoretical correctness in the data model to get the performance you need.

Before you make any changes, however, make sure that the queries are as efficient as possible. Study Chapter 4, "Optimizing Your Queries," to see what you can do to improve performance without changing the data model.

The remainder of this chapter contains discussions on the following topics:

- The disk storage methods used by **INFORMIX-OnLine** and their implications for performance and reliability

- The methods for calculating the sizes of tables and indexes, and using those numbers to estimate query-execution time

- The benefits and costs of adding indexes

- The ways you can "denormalize" the data model, that is, spoil its theoretical purity to improve performance

- Suggestions for increasing concurrency among multiple programs

INFORMIX-OnLine Disk Storage

INFORMIX-OnLine manages disk storage directly. To the greatest extent possible, it bypasses the host operating system and works with the raw disk space. (The extent to which this is possible depends on the operating system.)
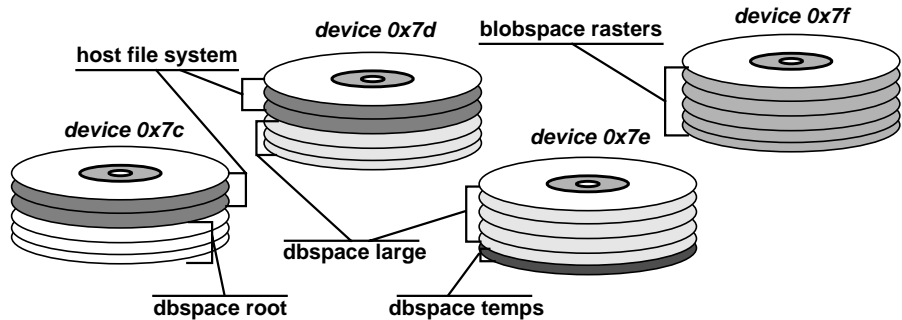
Figure 10-1   The relationship between chunks and dbspaces

Its methods of disk management are not complicated. However, several different terms are used for different amounts of space in different contexts: the *chunk*, the *dbspace* (or *blobspace*), the *page*, the *extent*, and the *tblspace*.

Chunks and Pages

The basic unit of physical storage is the *chunk*. A chunk is a unit of disk storage that is dedicated to **INFORMIX-OnLine**. Usually it is a physical device, that is, a disk drive. However, a chunk can be part of one disk, or simply a file. In any case, a chunk is a unit of space that can be identified to the operating system as being in the exclusive control of **INFORMIX-OnLine**.

The *page* is the basic unit of disk input/output (I/O). All space in every chunk is divided into pages. All I/O is done in units of whole pages. The size of a page is the same in all chunks used for tables. It is set when an **INFORMIX-OnLine** system is installed.

Dbspaces and Blobspaces

A *space* always comprises one or more complete chunks. Often a space contains exactly one chunk, so that the space and the chunk are identical. However, when a space comprises more than one chunk, **INFORMIX-OnLine** makes the multiple chunks appear to be a single sequence of pages.

A space is either a *dbspace* or a *blobspace*, depending on its use. If a space contains databases and tables, it is a dbspace. If it is dedicated to holding Binary Large Objects (the blob data types TEXT and BYTE), it is a blobspace. Chunks are assigned to a space when it is created. Chunks also can be added to a space later.

One dbspace, the *root* dbspace, is the first created; it always exists. It is the most important space because it holds the control information that describes all other spaces and chunks.

Parts of a single database can appear in two or more dbspaces. However, a single table is always completely contained in one dbspace.

Disk Mirroring

Individual spaces can be *mirrored*. The chunks of a mirrored space are paired with other chunks of equal size. Whenever a page is written to one of the chunks, it is also written to the mirror chunk. When one chunk suffers a hardware failure, the database server uses its mirror to continue processing without interruption. When the broken chunk is restored to service (or when a substitute chunk is assigned), **INFORMIX-OnLine** automatically brings it back to equality with the working chunk and continues operations.

When any dbspaces are mirrored, the root dbspace also should be mirrored. Otherwise, if it is lost to hardware failure, all **INFORMIX-OnLine** data is unusable regardless of mirrors.

If your database has extreme requirements for reliability in the face of hardware failure, you should arrange for it to be placed in a mirrored dbspace. As indicated in the following paragraphs, it is possible to locate individual tables in particular dbspaces. Hence, you can place some tables in mirrored spaces and other tables in normal spaces.

Databases

A database resides initially in one dbspace. It is placed there by a parameter of the CREATE DATABASE statement. The following example creates a database in the dbspace named **dev0x2d**:

```
CREATE DATABASE reliable IN dev0x2d WITH BUFFERED LOG
```

When no dbspace is specified, a database is placed in the root dbspace. When a database is in a dbspace it means only that

- Its system catalog tables are stored in that dbspace
- That dbspace is the default location for tables that are not explicitly created in other dbspaces

Tables and Spaces

A table resides completely in one dbspace. (Its blob values can reside in separate blobspaces or on an optical storage subsystem.) If no dbspace is specified, a table resides in the dbspace where its database resides. The following partial example creates a table in the dbspace named **misctabs**:

```
CREATE TABLE taxrates (…column specifications…) IN misctabs
```

You can achieve many different aims by placing a table in a specific dbspace. Some of these aims are explored in the following paragraphs.

Exploiting Mirroring

Place all the tables used by a critically important application in a mirrored dbspace. Alternatively, create the database in the mirrored dbspace and let the important tables reside there by default. Tables that are part of the database, but which are not used by the critical programs, can be located in non-mirrored spaces.

Sharing Temporary Space

When databases are large and disk space is in short supply, the normal dbspaces might not have enough room for those large temporary tables that can be helpful in improving query performance. However, a temporary table only exists for the life of the program that creates it—even less time if the program drops the table promptly. Set up a single dbspace for temporary tables. It can be used repeatedly by many programs.

Assigning Dedicated Hardware

A dbspace can equal a chunk, and a chunk can equal a device. Thus, you can place a table on a disk device that is dedicated to its use. When disk drives have different performance levels, you can put the tables with the highest frequency of use on the fastest drives.

By giving a high-use table a dedicated access arm, you can reduce contention with applications that use other tables. This does not reduce contention between programs using the same table unless the table itself is spread across multiple devices, as suggested in the next paragraph.

Assigning Additional Access Arms to a Table

A dbspace can comprise multiple chunks, and each chunk can represent a different disk. This permits you to assign multiple disk-access arms to one table. Figure 10-2 shows an example in graphical terms.
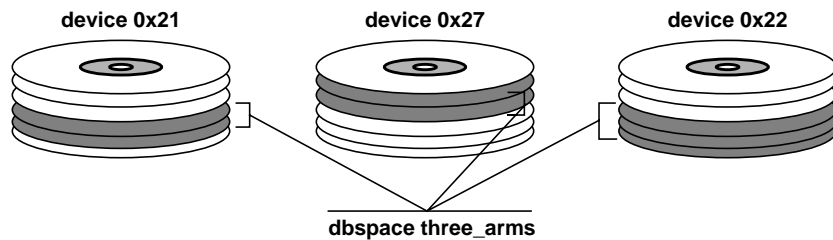
Figure 10-2   A dbspace distributed over three disks

The dbspace named **three_arms** comprises chunks on three different disks. Each chunk is approximately one-third the size of an important, heavily used table. (The third chunk is larger to allow for growth.) When multiple programs query the table, their I/O operations are distributed over the three disks, reducing contention for the use of the hardware.

In an arrangement like this, the multiple disks can act as mirrors for each other. For example, device **0x21** can be mirrored on **0x27**, **0x27** on **0x22**, and **0x22** on **0x21**. As long as any two devices are operational, the dbspace is usable. (However, if a single piece of hardware, such as a disk controller, is common to all three disks, this type of mirror arrangement is still vulnerable to failure of that component.)

Tblspaces

The total of all disk space allocated to a table is the *tblspace* of the table. The tblspace includes pages allocated to data (rows of that table) and pages allocated to indexes. It also includes pages used by blob columns that are located in the tblspace, but not pages used by blob data in a separate blobspace or blob data stored on an optical storage subsystem. (This choice is discussed under "Locating Blob Data" on page 10-18.)

The tblspace is an accounting entity only; it does not correspond to any particular part of a dbspace. The indexes and data extents that make up a table can be scattered throughout the dbspace.

The **tbcheck** utility (with the -**pt** option) returns information on the status of tblspaces, including the counts of pages allocated and in use for each one. You can use this information to monitor the growth of a table over time.

Extents

As you add rows to a table, **INFORMIX-OnLine** allocates disk space to it in units called *extents*. Each extent is a block of physically contiguous pages from the dbspace. Even when the dbspace comprises more than one chunk, extents are always allocated entirely within a single chunk to remain contiguous.

Contiguity is important to performance. When the pages of data are contiguous, disk-arm motion is minimized when the database server reads the rows sequentially. The mechanism of extents is a compromise between the following competing requirements:

- Most dbspaces are shared among several tables.
- The size of some tables is not known in advance.
- Tables can grow at different times and different rates.
- All the pages of a table should be adjacent for best performance.

Since table sizes are not known, table space cannot be preallocated. Therefore, extents are added only as they are needed, but all the pages in any one extent are contiguous for better performance.

Choosing Extent Sizes

When you create a table, you can specify the size of the first extent as well as the size of the extents to be added as the table grows. You can change the size of added extents later with the ALTER TABLE statement. The following partial example creates a table with a half-megabyte initial extent and 100-kilobyte added extents:

```
CREATE TABLE big_one (…column specifications…)
    IN big_space
    EXTENT SIZE 512
    NEXT SIZE 100
```

The following example changes the next-extent size of the table to 50 kilo-bytes. This has no effect on extents that already exist.

```
ALTER TABLE big_one MODIFY NEXT SIZE 50
```

The next-extent sizes of the following kinds of tables are not very important to performance:

- A small table has only one extent (otherwise it would not be small). If it is heavily used, large parts of it are buffered in memory anyway.

- An infrequently used table is not important to performance no matter what size it is.

- A table that resides in a dedicated dbspace always receives new extents that are adjacent to its old extents. The size of these extents is not important because, being adjacent, they perform like one large extent.

When you assign an extent size to these kinds of tables, the only consideration is to avoid creating large numbers of extents. A large number of extents causes the database server to spend a small amount of extra time on book-keeping. Also, there is an upper limit on the number of extents allowed. (This is covered in the section "Upper Limit on Extents" on page 10-10.)

Next-extent sizes become important when two or more large and growing tables share a dbspace. Because the extents added to the different tables are interleaved, each new extent represents another long seek that must be made when reading sequentially. It also extends the total distance over which the disk arm must range when reading nonsequentially.

There is no upper limit on extent sizes except the size of the dbspace. When you know the final size of a table (or confidently can predict it within 25%), allocate all its space in the initial extent. When tables grow steadily to unknown size, assign them next-extent sizes that let them share the dbspace with a small number of extents each. Here is one possible approach:

- Decide on the ratio in which the dbspace is shared between the tables. For example, you might divide the dbspace among three tables in the ratio 0.4 : 0.2 : 0.3 (reserving 10% for small tables and overhead).

- Give each table one-fourth of its share of the dbspace as its initial extent.

- Assign each table one-eighth of its share as its next-extent size.

Monitor the growth of the tables regularly using **tbcheck**.

What happens if, as the dbspace fills up, there is not enough contiguous space to create an extent of the size you specified? In that case, **INFOR-MIX-OnLine** allocates the largest contiguous extent that it can.

Upper Limit on Extents

No table should be allowed to acquire a large number of extents. But it can happen, and there is an upper limit on the number of extents allowed. Trying to add an extent after the limit is reached causes **ISAM** error -136 (`No more extents`) to follow an INSERT request.

The upper limit depends on the page size and the table definition. To learn the upper limit on extents for a particular table, calculate a series of values as follows:

pagesize =	the size of a page (reported by **tbstat** -**c** under the head BUFFSIZE)
colspace =	4 × the number of columns in the table
ixspace =	12 × the number of indexes on the table
ixparts =	4 × the number of columns named in each index
extspace =	*pagesize* - (*colspace* + *ixspace* + *ixparts* + 84)

The table can have no more than *extspace*/8 extents. When the page size is 2,048, the **customer** table from the demonstration database can have no more than 234 extents, as calculated in Figure 10-3.

To help ensure that the limit is not exceeded, **INFORMIX-OnLine** checks the number of extents each time it creates a new one. If the extent being created is the 64th or a multiple of 64, the next-extent size for the table is automatically doubled.

Variable	Value	Basis
pagesize	2,048	output of **tbstat**
colspace	40	ten columns
ixspace	36	three indexes
ixparts	12	one column in each index
extspace	1,876	2,048 - (40 + 36 + 12 + 84)
limit	234	*extspace*/8

Figure 10-3   ***Values used to determine upper limit on extents***

Defragmenting Tables

Query performance can suffer after many extents are added to a table. This can only be the case when two or more large, growing tables share a dbspace. When tables grow at the same time, their new extents and index pages are interleaved, creating large gaps between the extents of any one table. This situation is diagrammed in Figure 10-4.

Figure 10-4   A fragmented dbspace

The figure also shows gaps of unused space, perhaps created by tables that existed when extents were allocated but which have since been dropped.

Disorganization such as that depicted in Figure 10-4 hurts performance in two ways. During sequential access to any table, several long seeks must be made. For nonsequential access to any table, the disk might have to seek across the entire width of the dbspace. It is possible to rebuild a dbspace so that tables are compact once again, as shown in Figure 10-5. The relative order of the reorganized tables within the dbspace is not important; all that matters is that the pages of each are together. When reading a table sequentially, there are no long seeks. When reading a table nonsequentially, the disk arm ranges only over the space occupied by that table.

Figure 10-5   A dbspace reorganized to make tables compact

Use the following steps to reorganize a dbspace:

- Copy the tables in the dbspace to tape individually, using the **tbunload** utility.
- Drop all the tables in the dbspace.
- Re-create the tables using the **tbload** utility.

Th**e tbload** utility re-creates the tables with the identical properties they had before, including the same extent sizes. When a new extent is created adjacent to the previous extent, the two are treated as a single extent.

You can also unload a table using the UNLOAD command of **DB**-**Access**, **INFORMIX**-**SQL**, or **INFORMIX**-**4GL**, and reload the table using the companion LOAD command or the **dbload** utility. However, these operations convert the table into character form, while **tbload** and **tbunload** work with binary copies of disk pages.

There are two more ways to reorganize a single table. If you use the ALTER TABLE command to add or drop a column or to change the data type of a column, the table is copied and reconstructed. If you create a clustered index or alter an index to cluster, the table is sorted and rewritten. (See "Clustered Indexes" on page 10-24.) In both cases, the table is written on other areas of the dbspace. However, if other tables are in the dbspace, there is no guarantee that all the new extents are adjacent.

Calculating Table Sizes

This section discusses methods for calculating the approximate sizes of tables and indexes in disk pages. Like the previous sections, it applies only to **INFORMIX**-**OnLine**. Estimates of this kind are useful when you are planning the layout of a database on disk.

These calculations aim to estimate the number of disk pages that are used to hold different parts of a table. If the table already exists, or if you can build a demonstration table of realistic size using simulated data, you do not have to make estimates. You can run **tbcheck** and obtain exact numbers.

You can calculate a series of values to produce the estimates. You need the following values in all estimates. Let the values be as follows:

estrows = the number of rows you estimate the table has when it reaches its normal size.

pagesize = the size of a page (reported by **tbstat** -**c** under the head BUFFSIZE).

pageuse = *pagesize* - 28 (the space available for data on a disk page).

In the calculations in this chapter, floor(*x*) means the largest integer smaller than *x*, while ceiling(*x*) means the smallest integer larger than *x*.

Estimating Fixed-Length Rows

When a table contains no VARCHAR columns, its rows all have the same size. Calculate the following additional variables. Let the values be as follows:

rowsize = the sum of the sizes of all columns in the table, plus 4 bytes. Sizes of different data types are discussed in Chapter 9. Treat TEXT and BYTE columns as having size 56.

homerow = `if` (*rowsize* ≤ *pageuse*) then *rowsize*
else 4 + remainder (*rowsize*/*pageuse*)

If a row is larger than a page, as many full pages as possible are removed from the end of the row and stored in separate expansion pages. Let *homerow* be the size of the leading part, which is kept on the home page. The value *homerow* is simply *rowsize* if a row fits a page.

overpage = `if` (*rowsize* ≤ *pageuse*) then 0
else floor (*rowsize*/*pageuse*)

Let *overpage* be the number of expansion pages needed for each row; it is zero when a row fits on a page.

datrows = min(255,floor (*pageuse*/*homerow*))

Let *datrows* be the number of rows (or leading parts of rows) that fits on one page. There is an upper limit of 255 rows per page, even when rows are very short.

datpages = ceiling (*estrows*/*datrows*)

expages = *estrows* × *overpage*

The table requires approximately *datpages* + *expages* disk pages for rows. (Additional pages for TEXT and BYTE columns are estimated in "Estimating Blobpages" on page 10-17.) An estimate for the **customer** table is shown in Figure 10-6.

Variable	Estimate	Basis of Estimate		
estrows	1,500	company business plan		
pagesize	2,048	output of **tbstat**		
pageuse	2,020	*pagesize* - 28		
rowsize	138	sum of 4+ **column**	**data type**	**size**
		customer_num	SERIAL	4
		fname	CHAR(15)	15
		lname	CHAR(15)	15
		company	CHAR(20)	20
		address1	CHAR(20)	20
		address2	CHAR(20)	20
		city	CHAR(15)	15
		state	CHAR(2)	2
		zipcode	CHAR(5)	5
		phone	CHAR(18)	18
homerow	134	*rowsize* ≤ *pageuse*		
overpage	0	*rowsize* ≤ *pageuse*		
datrows	14	floor (*pageuse*/ *homerow*)		
datpages	108	ceiling (*estrows*/ *datrows*)		
expages	0	*estrows* × *overpage*		

Figure 10-6   *Estimating the size of the* **customer** *table*

Estimating Variable-Length Rows

When a table contains one or more VARCHAR columns, its rows have varying lengths. This introduces uncertainty. You must form an estimate of the typical size of each VARCHAR column, based on your understanding of the data.

When **INFORMIX-OnLine** allocates space to rows of varying size, it considers a page to be full when there is not room for an additional row of the *maximum* size. This can limit the allocation of rows to pages. Calculate the following variables:

maxrow = the sum of the maximum sizes of all columns in the table, plus 4 bytes. Treat TEXT and BYTE columns as having size 56.

typrow = the sum of the estimated typical sizes of all columns in the table, plus 4 bytes.

homerow = if (*typrow* ≤ *pageuse*) then *typrow*
else 4 + remainder (*typrow*/ *pageuse*)

Let *homerow* be the amount of a typical row that fits on the home page; it is *typrow* if one page is sufficient.

$overpage =$ `if` ($typrow \leq pageuse$) then 0
                       else floor ($typrow / pageuse$)

                       Let *overpage* be the number of expansion pages needed for the typical row; it is zero when a typical row fits on a page.

$homemax =$ if ($maxrow \leq pageuse$) then *maxrow*
                       else 4 + remainder ($maxrow / pageuse$)

                       Let *homemax* be the reserve requirement of the database server.

$datrows =$ floor (($pageuse - homemax$) / $typrow$)

                       Let *datrows* be the number of typical rows that fits on a page before the database server decides the page is full.

$datpages =$ ceiling ($estrows / datrows$)

$expages =$ $estrows \times overpage$

The table requires approximately *datpages* + *expages* disk pages for rows. (Additional pages for TEXT and BYTE columns are estimated in "Estimating Blobpages" on page 10-17.) An estimate for the **catalog** table is shown in Figure 10-6.

Variable	Estimate	Basis of Estimate			
estrows	5,000	company business plan.			
pagesize	2,048	output of **tbstat**			
pageuse	2,020	*pagesize* -28			
maxrow	381	sum of 4+:			
		column	data type	max size	typical
		catalog_num	SERIAL	4	4
		stock_num	SMALLINT	2	2
		manu_code	CHAR(3)	3	3
		cat_descr	TEXT	56	56
		cat_picture	BYTE	56	56
		cat_adv	VARCHAR(255,65)	256	66
typrow	191				
homerow	191	$typrow \leq pageuse$			
overpage	0	$typrow \leq pageuse$			
homemax	381	$maxrow \leq pageuse$			
datrows	8	floor (($pageuse - homemax$) / $typrow$)			
datpages	625	ceiling ($estrows / datrows$)			
expages	0	$estrows \times overpage$			

Figure 10-7   ***Estimating the size of the* catalog *table***

Estimating Index Pages

The tblspace includes index pages as well as pages of data, and index pages can be a significant fraction of the total.

An index entry consists of a *key* and a *pointer*. The key is a copy of the indexed column or columns from one row of data. The pointer is a 4-byte value used to locate the row containing that key value.

A unique index contains one such entry for every row in the table. It also contains some additional pages of pointers that create the B+ tree structure. (Index structure is described in "The Structure of an Index" on page 4-26.) When an index allows duplicates, it contains fewer keys than pointers; that is, one key can be followed by a list of pointers.

If this were the whole story on indexes, it would be simple to estimate their space requirements. However, the index leaf pages use a technique called *key compression,* which allows the database server to put more keys on a page than would otherwise fit. The space saved by key compression varies with the content of the keys, from page to page and index to index.

To estimate the size of a compact index (disregarding space saved by key compression), let the values be as follows:

keysize = the total width of the indexed column or columns.

pctuniq = the number of unique entries you expect in this index, divided by *estrows*.

For a unique index, or one with only occasional duplicate values, use 1.0. When duplicates are present in significant numbers, let *pctuniq* be a fraction less than 1.0.

entsize = ($keysize \times pctuniq$) + 4

pagents = floor (*pageuse/entsize*)

There are approximately *pagents* entries on each index page.

leaves = ceiling (*estrows/pagents*)

There are approximately *leaves* leaf pages in the index.

branches = ceiling (*leaves/pagents*)

There are approximately *branches* nonleaf pages.

The index contains approximately *leaves + branches* pages when it is compact. Estimates for two indexes in the **customer** table are carried out in Figure 10-8 on page 10-17.

As rows are deleted and new ones inserted, the index can become sparse; that is, the leaf pages might no longer be full of entries. On the other hand, if key compression is effective, the index might be smaller. The method given here should yield a conservative estimate for most indexes. If index space is important, build a large test index using real data and check its size with the **tbcheck** utility.

Variable	Estimate	Basis for Estimate
estrows	1,500	company business plan
pagesize	2,048	output of **tbstat**
pageuse	2,016	*pagesize*
keysize	4	customer_num is 4 bytes
pctuniq	1.0	unique index
entsize	12	4 x 1.0 + 8
pagents	168	floor (2,016/12)
leaves	9	ceiling (1,500/168)
branches	1	ceiling (9/168)
keysize	2	state is CHAR(2)
pctuniq	0.033	50 states among 1,500 rows
entsize	8.066	2 x 0.033 + 8
pagents	249	floor (2,016/8.066)
leaves	7	ceiling (1,500/249)
branches	1	ceiling (7/168)

Figure 10-8   Estimating the size of two indexes from the customer *table*

Estimating Blobpages

BYTE and TEXT data items that are stored on magnetic disk are stored in separate pages, either interspersed among the row and index pages in the tblspace or in a separate blobspace. (This discussion does not apply to blob data stored on optical media.) Each blob data item occupies a whole number of pages. For each blob column, let the values be as follows:

typage = the number of whole pages required to store an item of typical size.

nnpart = the fraction of rows in which this column has a non-null value.

totpage= $typage \times estrows \times nnpart.$

The values for the column occupy approximately *totpage* pages. In the **catalog** table (see Figure 10-6 on page 10-14) there are two blobpages. Their estimates can be imagined as follows:

For the **cat_descr** column, the text of a description is, at most, a double-spaced page, which is 250 words or approximately 1,500 characters, so *typage* = 1. There is a description for every entry, so *nnpart* = 1.0. Thus, *totpage* = $1 \times 5,000 \times 1.0 = 5,000$ pages of data.

The **cat_picture** column contains line art in a computer-readable form. Examination of a few of these pictures reveals that they vary widely in size but a typical file contains about 75,000 bytes of data. Thus *typage* = 38. The marketing department estimates that they want to store a picture with one entry in four: *nnpart* = 0.25. Therefore *totpage* = $38 \times 5,000 \times 0.25 = 47,500$ pages of data.

After a table is built and loaded, you can check the usage of blobpages with the **tbcheck** utility -**ptT** option.

Locating Blob Data

When you create a column of type BYTE or TEXT on magnetic disk, you have the option of locating the data of the column in the tblspace or in a blobspace. (This discussion does not apply to blob data stored on optical media.) In the following example, a TEXT value is located in the tblspace and a BYTE value is located in a blobspace named **rasters**.

```
CREATE TABLE examptab
    (
    pic_id SERIAL,
    pic_desc TEXT IN TABLE,
    pic_raster BYTE IN rasters
    )
```

A TEXT or BYTE value is always stored apart from the rows of the table. Only a 56-byte descriptor is stored with the row. However, the value itself occupies at least one disk page.

When blob values are stored in the tblspace, the pages of their data are interspersed among the pages containing rows. The result is to inflate the size of the table. The blobpages separate the pages containing rows and spread them out on the disk. When the database server reads only the rows and not the blob data, the disk arm must move farther than it would if the blobpages were stored apart. The database server scans only the row pages on any SELECT operation that retrieves no blob column, and whenever it tests rows using a filter expression.

Another consideration is that disk I/O to and from a dbspace is buffered. Pages are held in storage in case they are needed again soon; and, when pages are written, the requesting program is allowed to continue before the actual disk write takes place.

However, disk I/O to and from blobspaces is not buffered. Blobspace pages are not retained in buffers to be read again, and the requesting program is not allowed to proceed until all output to them is complete. The reason is that blobspace input and output are expected to be voluminous. If these pages are passed through the normal buffering mechanisms, they could monopolize the buffers, driving out index pages and other pages useful to good performance.

For best performance, then, you should locate a TEXT or BYTE column in a blobspace in either of these circumstances:

- When single data items are larger than one or two pages each; if kept in the dbspace, their transfer dilutes the effectiveness of the page buffers.

- When the number of pages of blob data is more than half the number of pages of row data; if kept in the dbspace, the table is inflated and queries against it are slowed.

For a table that is both relatively small and nonvolatile, you can achieve the effect of a dedicated blobspace by the following means: Load the entire table with rows in which the blob columns are null. Create all indexes. Row pages and index pages are now contiguous. Update all the rows to install the blob data. The blob pages follow the pages of row and index data in the tblspace.

Managing Indexes

An index is necessary on any column (or composition of columns) that must be unique. However, as discussed in Chapter 4, the presence of an index can also allow the query optimizer to speed up a query in several ways. The optimizer can use an index as follows:

- To replace repeated sequential scans of a table with nonsequential access

- To avoid reading row data at all when processing expressions that name only indexed columns

- To avoid a sort (including building a temporary table) when executing the GROUP BY and ORDER BY clauses

As a result, an index on the right column can save thousands, tens of thousands, or in extreme cases even millions of disk operations during a query. However, there are costs associated with indexes.

Space Costs of Indexes

The first cost of an index is disk space. An estimating method is given in "Estimating Index Pages" on page 10-16. Loosely, an index contains a copy of every unique data value in the indexed columns, plus a 4-byte pointer for every row in the table. This information can add many pages to the table space requirements; it is easy to have as many index pages as row pages.

Time Costs of Indexes

The second cost of an index is time while the table is modified. The descriptions that follow assume that approximately two pages must be read to locate an index entry. That is the case when the index consists of a root page, one intermediate level, and leaf pages, and the root page is already in a buffer. The index for a very large table has two intermediate levels, so roughly three pages are read when an entry is looked up.

Presumably, one index has been used to locate the row being altered. Its index pages are found in the page buffers. However, the pages for any other indexes that need altering must be read from disk.

Under these assumptions, index maintenance adds time to different kinds of modifications as follows:

- When a row is deleted from a table, its entries must be deleted from all indexes.

  The entry for the deleted row must be looked up (two or three pages in) and the leaf page must be rewritten (one page out), for a total of three or four page accesses per index.

- When a row is inserted, its entries must be inserted in all indexes.

  The place for entry of the inserted row must be found (two or three pages in) and rewritten (one page out), for a total of three or four page accesses per index.

- When a row is updated, its entries must be looked up in each index that applies to a column that was altered (two or three pages in). The leaf page must be rewritten to eliminate the old entry (one page out); then the new column value must be located in the same index (two or three more pages in) and the row entered (one more page out).

Insertions and deletions change the number of entries on a leaf page. In virtually every *pagents* operation, some additional work must be done because a leaf page has either filled up or emptied. However, since *pagents* is usually greater than 100, this occurs less than 1% of the time and can be disregarded for estimating.

In short, when a row is inserted or deleted at random, allow three to four added page I/O operations per index. When a row is updated, allow six to eight page I/O operations for each index that applies to an altered column. Bear in mind also that if a transaction is rolled back, all this work must be undone. For this reason, rolling back a transaction can take a long time.

Since the alteration of the row itself requires only two page I/O operations, it is clear that index maintenance is the most time-consuming part of data modification. One way to reduce this cost is discussed under "Dropping Indexes" on page 10-23.

Choosing Indexes

Indexes are required on columns that must be unique and are not specified as primary keys. In addition, you should add an index in three other cases:

- Columns used in joins that are not specified as foreign keys
- Columns frequently used in filter expressions
- Columns frequently used for ordering or grouping

Join Columns

As discussed in Chapter 4, at least one column named in any join expression should have an index. If there is no index, the database server usually builds a temporary index before the join and discards it afterward; that is almost always faster than performing a join by repeated sequential scans over a table.

When both columns in a join expression have indexes, the optimizer has more options when it constructs the query plan. As a general rule, you should put an index on any column that is used in a join expression more than occasionally that is not identified as a primary or foreign key.

Selective Filter Columns in Large Tables

If a column is often used to filter the rows of a large table, consider placing an index on it. The optimizer can use the index to pick out the desired columns, avoiding a sequential scan of the entire table. One example might be a table

that contains a large mailing list. If you find that a postal-code column is often used to filter a subset of rows, you should consider putting an index on it even though it is not used in joins.

This strategy yields a net savings of time only when the selectivity of the column is high; that is, when only a small fraction of rows holds any one indexed value. Nonsequential access through an index takes several more disk I/O operations than sequential access, so if a filter expression on the column passes more than a fourth of the rows, the database server might as well read the table sequentially. As a rule, indexing a filter column saves time in the following cases:

- The column is used in filter expressions in many queries or in slow queries.
- The column contains at least 100 unique values.
- Most column values appear in fewer than 10% of the rows.

Order-By and Group-By Columns

When a large quantity of rows must be ordered or grouped, the database server must put the rows in order. One way the database server does this is to select all the rows into a temporary table and sort the table. But (as discussed in Chapter 4) if the ordering columns are indexed, the optimizer sometimes plans to read the rows in sorted order through the index, thus avoiding a final sort.

Since the keys in an index are in sorted sequence, the index really represents the result of sorting the table. By placing an index on the ordering column or columns, you can replace many sorts during queries with a single sort when the index is created.

Duplicate Keys Slow Index Modifications

When duplicate keys are permitted in an index, the entries that have any single value are grouped in a list. When the selectivity of the column is high, these lists are generally short. But when there are only a few unique values, the lists become quite long and, in fact, can cross multiple leaf pages.

For example, in an index on a column whose only values are M for married and S for single, all the index entries are contained in just two lists of duplicates. Such an index is not of much use, but at least it works for querying; the database server can read out the list of rows that have one value or the other.

When an entry must be deleted from a list of duplicates, the database server must read the whole list and rewrite some part of it. When it adds an entry, the database server puts the new row at the end of its list. Neither operation is a problem when the list is short, as is normal. But when a list fills many pages, the database server must read all the rows to find the end. When it deletes an entry, it typically must update and rewrite half the pages in the list.

Thus, an index on a column that has a small number of distinct values, in a table that has a large number of rows, can drastically reduce the speed of updating. An example is a column whose values are the names or abbreviations of states or provinces. If there are 50 unique values in a mailing list of 100,000 rows, there are an average of 2,000 duplicates per value. But real data is never so well distributed; in such a table the more common values likely have 10,000 or more duplicates, and their lists might approach 50 pages in length.

When the database server inserts or deletes an entry in such a list, it is busy for a long time. Worse still, it has to lock all the affected index pages while it does the work, greatly reducing concurrent access to the table.

You can avoid this problem in a fairly simple way at some cost in disk space. The trick is to know that the database server uses the leading column of a composite index in the same way as it uses an index on that column alone. So instead of creating an index on a column with few unique values, create a composite index on that column followed by one other column that has a wide distribution of values.

For example, change the index on the column whose only values are M and S into a composite index on that column and a birthdate column. You can use any second column to disperse the key values as long as its value does not change, or changes at the same time as the real key. The shorter the second column the better, since its values are copied into the index and expand its size.

Dropping Indexes

In some applications, the majority of table updates can be confined to a single time period. Perhaps all updates are applied overnight or on specified dates.

When this is the case, consider dropping all non-unique indexes while updates are being performed, then creating new indexes afterward. This can have two good effects.

First, since there are fewer indexes to update, the updating program can run faster. Often, the total time to drop the indexes, update without them, and re-create them afterward is less than the time to update with the indexes in place. (The time cost of updating indexes is discussed under "Time Costs of Indexes" on page 10-20.)

Second, newly made indexes are the most efficient ones. Frequent updates tend to dilute the index structure, causing it to contain many partly full leaf pages. This reduces the effectiveness of an index and wastes disk space.

As another timesaving measure, make sure that a batch-updating program calls for rows in the sequence defined by the primary-key index. That sequence causes the pages of the primary-key index to be read in order and only one time each.

The presence of indexes also slows down the population of tables when you use the LOAD command or the **dbload** utility. Loading a table that has no indexes at all is a very quick process (little more than a disk-to-disk sequential copy), but updating indexes adds a great deal of overhead.

The fastest way to load a table is as follows:

1. Drop the table (if it exists).
2. Create the table without specifying any unique constraints.
3. Load all rows into the table.
4. Alter the table to apply the unique constraints.
5. Create the non-unique indexes.

If you cannot guarantee that the loaded data satisfies all unique constraints, you must create the unique indexes before you load the rows. It saves time if the rows are presented in the correct sequence for at least one of the indexes (if you have a choice, make it the one with the largest key). This minimizes the number of leaf pages that must be read and written.

Clustered Indexes

The term *clustered index* is a misnomer. There is nothing special about the index; it is the table that is modified so that its rows are physically ordered to agree with the sequence of entries in the index. (Do not confuse the clustered index with an *optical cluster*, which is a method for storing logically related blobs together on an optical volume. For information on optical clusters, see the *INFORMIX-OnLine/Optical User Manual.*)

When you know that a table is ordered by a certain index, you can take advantage of the knowledge to avoid sorting. You can also be sure that when the table is searched on that column, it is read (effectively) in sequential order instead of nonsequentially. These points are covered in Chapter 4.

In the **stores5** database, the **orders** table has an index, **zip_ix,** on the zip code column. The following command causes the database server to put the rows of the **customer** table into descending order by zip code:

```
ALTER INDEX zip_ix TO CLUSTER
```

To cluster a table on a nonindexed column, you must create an index. The following command reorders the **orders** table by order date:

```
CREATE CLUSTERED INDEX o_date_ix ON orders (order_date ASC)
```

To reorder a table, the database server must copy the table. In the preceding example, the database server reads all rows of the table and constructs an index. Then it reads the index entries in sequence. For each entry, it reads the matching row of the table and copies it to a new table. The rows of the new table are in the desired sequence. This new table replaces the old table.

Clustering is not preserved when you alter a table. When you insert new rows, they are stored physically at the end of the table regardless of their contents. When you update rows and change the value of the clustering column, the rows are nevertheless written back into their original location in the table.

When clustering is disturbed by updates, it can be restored. The following command reorders the table to restore the physical sequence:

```
ALTER INDEX o_date_ix TO CLUSTER
```

Reclustering is usually quicker than the original clustering because reading out the rows of a nearly clustered table is almost a sequential scan.

Clustering and reclustering take a lot of space and time. You can avoid some clustering by building the table in the desired order in the first place. The physical order of rows is their insertion order, so if the table is initially loaded with ordered data, no clustering is needed.

Denormalizing

Extended Relational Analysis, the method of data modeling described in Chapter 8, produces tables that contain no redundant or derived data; tables that are well-structured by the tenets of relational theory.

Sometimes, to meet extraordinary demands for high performance, you might have to modify the data model in ways that are undesirable from a theoretical standpoint. This section describes some modifications and their associated costs.

Shorter Rows for Faster Queries

As a general principle, tables with shorter rows yield better performance than ones with longer rows. This is because disk I/O is performed in pages, not in rows. The shorter the rows of a table, the more rows there are on a page. The more rows per page, the fewer I/O operations it takes to read the table sequentially, and the more likely it is that a nonsequential access can be satisfied from a buffer.

Extended Relational Analysis had you put all the attributes of one entity into a single table for that entity. For some entities, this can produce rows of awkward length. There are some ways to shorten them. As the rows get shorter, query performance should improve.

Expelling Long Strings

The most bulky attributes are often character strings. If you can remove them from the entity table, the rows become shorter.

Using VARCHAR Strings

Since the VARCHAR data type is relatively new with **INFORMIX-OnLine**, an existing database might contain CHAR columns that can be converted profitably to VARCHAR. VARCHAR columns shorten the average row when the average value in the CHAR column is at least 2 bytes shorter than the existing, fixed width of the column.

This substitution does not harm the theoretical characteristics of the model. Furthermore, VARCHAR data is immediately compatible with most existing programs, forms, and reports. (Forms must be recompiled. Forms and reports should, of course, be tested on a sample database.)

Changing Long Strings to TEXT

When the typical string fills half a disk page or more, consider converting it to a TEXT column in a separate blobspace. The column within the row page is only 56 bytes long, which should put many more rows on a page. However, the TEXT data type is not automatically compatible with existing programs. The code for fetching a TEXT value is more complicated than the code for fetching a CHARACTER value into a program.

Building a Symbol Table of Repeated Strings

If a column contains strings that are not unique in each row, you can remove those strings to a table in which only unique copies are stored.

For example, the **customer.city** column contains city names. Some city names are repeated down the column, and most rows have some trailing blanks in the field. Using the VARCHAR data type eliminates the blanks but not the duplication.

You can create a table named **cities**, as follows:

```
CREATE TABLE cities
    (
    city_num SERIAL PRIMARY KEY,
    city_name VARCHAR(40) UNIQUE
    )
```

Then you can change the definition of the **customer** table so that its **city** column becomes a foreign key that references the **city_num** column in the **cities** table.

You must change any program that inserts a new row into **customer** to insert the city of the new customer into **cities**. The database server return code in the SQLCODE field can indicate that the insert failed because of a duplicate key. It is not a logical error; it simply means that some existing customer is located in that city. (However, a **4GL** program must use the WHENEVER command to trap errors; otherwise the negative value in SQLCODE terminates the program.)

Besides changing programs that insert data, you also must change all programs and stored queries that retrieve the city name. They must use a join into the new **cities** table to obtain their data. The extra complexity in programs that insert rows and the extra complexity in some queries is the cost of giving up theoretical correctness in the data model. Before you make the change, be sure it returns a reasonable savings in disk space or execution time.

Moving Strings to a Companion Table

Strings less than half a page long waste disk space if you treat them as TEXT, but you can remove them from the main table to a companion table. The use of companion tables is the subject of the next section.

Splitting Wide Tables

Consider all the attributes of an entity whose rows are too wide for good performance. Look for some theme or principle on which they can be divided into two groups. Split the table into two, a primary table and a companion table, repeating the primary key in each one. The shorter rows allow each table to be queried or updated more quickly.

Division by Bulk

One principle on which you can divide an entity table is bulk: move the bulky attributes, which are usually character strings, to the companion table. Keep the numeric and other small attributes in the primary table. In the demonstration database, you can split the **ship_instruct** column from the **orders** table. You can call the companion table **orders_ship**. It has two columns, a primary key that is a copy of **orders.order_num** and the original **ship_instruct** column.

Division by Frequency of Use

Another principle for division of an entity is frequency of use. If a few attributes are rarely queried, they can be moved out to a companion table. In the demonstration database, it could be that the **ship_instruct**, **ship_weight**, and **ship_charge** columns are queried only in one program. In that case, you can move them out to a companion table.

Division by Frequency of Update

Updates take longer than queries, and the updating programs lock index pages and rows of data, slowing down the programs that only query. If certain attributes of an entity are updated frequently, while many others are changed only rarely or not at all—*and* if many queries select the latter and do not need the former—you can split the volatile columns to a companion table. The updating programs contend for the use of that table, while the query programs use the primary table.

Costs of Companion Tables

Splitting a table consumes extra disk space and adds complexity. There are two copies of the primary key for each row, one copy in each table. There are also two primary-key indexes. You can use the methods described in earlier sections to estimate the number of pages added.

You must modify existing programs, reports, and forms that use SELECT * because fewer columns are returned. Programs, reports, and forms that use attributes from both tables must perform a join to bring them together.

When you insert or delete a row, two tables must be altered instead of one. If you do not coordinate the alteration of the two tables (by making them within a single transaction, for example), semantic integrity is lost.

Splitting Tall Tables

A large table poses large management problems. The 400-megabyte table takes a long time to query, of course, but it also takes a long time to back up or restore. Sorting becomes radically slower because the sort goes to three or more merge levels; sorting can even become impossible because too much disk space is required. Indexes become less effective because the index structure has three or more intermediate levels.

Consider splitting such a table into segments. Each segment has the same arrangement of columns, indexes, and constraints, but a different subset of rows. The division into groups of rows should be based on the value of an attribute that appears in most queries (probably the primary key) so that a program or user can easily tell in which subtable a row belongs. For example, if the primary key is a number, rows can be allocated to ten subtables based on a middle digit.

If rows include a time or date attribute, it might be useful to split a table into segments by age, especially if recent rows are used more often than older ones.

The benefits of splitting a large table are that the segments can be treated separately for copying, sorting, indexing, archiving, and restoring. All these operations become easier. The segments can be assigned to tblspaces in different ways to make the best use of disk devices. And, as long as most queries are based on the attribute used for segmenting, queries should be dramatically faster.

The drawback of splitting a table is that operations that apply to the whole table become much more complicated. If a query cannot be directed to one subtable, it must be written in the form of a UNION of many subqueries, each on a different subtable. This greatly complicates the design of reports and browsing programs.

Redundant and Derived Data

The data model produced by the methods of Chapter 8 contains no redundant data (every attribute appears in just one table) and no derived data (data that can be computed from existing attributes is selected as an expression based on those attributes).

These features minimize the amount of disk space used and make updating the tables as easy as possible. However, they can force you to use joins and aggregate functions often, and that may take more time than is acceptable.

As an alternative, you can introduce new columns that contain redundant or derived data, provided you understand the risks.

Adding Derived Data

In the **stores5** database, there is no column in the **orders** table for the total price of an order. The reason is that the information can be derived from the rows of the **items** table. A program that wants to report the date and total price of order number 1009 can obtain it with the following query:

```
SELECT order_date, SUM (total_price)
    FROM orders, items
    WHERE orders.order_num = 1009
        AND orders.order_num = items.order_num
    GROUP BY orders.order_num, orders.order_date
```

While the join to the **items** table only reads three or four additional pages, that might be too much time for an interactive program. One solution is to add an order-total column to the **orders** table.

The costs of derived data are in disk space, complexity, and data integrity.

The disk space devoted to an order-total column is wasted because the same information is stored twice. Also, the presence of the column makes the rows of the table wider; there are fewer of them in a page and querying the table becomes slower. Most important, any program that updates the base

attributes must be changed to also update the derived column. Inevitably, there are times when the derived column is wrong, when it does not contain the right derived value.

Derived data is not reliable data. In the example of a derived order price, the unreliability occurs while an order is being entered. While rows are being added to the **items** table, there are times when the order total in **orders** is not equal to the sum of the corresponding rows from **items**. In general, before you allow derived data to appear in the database, you must define its accuracy as carefully as possible. Actually, you should define as closely as possible the conditions under which the derived column is *un*reliable.

Adding Redundant Data

A correct data model avoids redundancy by keeping any attribute only in the table for the entity it describes. If the attribute data is needed in a different context, you make the connection by joining tables. But joining takes time. If a frequently used join hurts performance, you can eliminate it by duplicating the joined data in another table.

In the demonstration database, the **manufact** table contains the names of manufacturers and their delivery times. (In a real database, it would contain many other attributes of a supplier, such as address, sale representative name, and so on.)

The contents of **manufact** are primarily a supplement to the **stock** table. Suppose that a time-critical application frequently refers to the delivery lead time of a particular product, but to no other column of **manufact**. For each such reference, the database server must read two or three pages of data to perform the lookup.

You can add a new column, **lead_time**, to the **stock** table and fill it with copies of the **lead_time** column from the corresponding rows of **manufact**. That eliminates the lookup, speeding the application.

Like derived data, redundant data takes space and poses an integrity risk. In the example, there is not one extra copy of each manufacturer's lead-time, there are many. (Each manufacturer can appear in **stock** many times.) The programs that insert or update a row of **manufact** must be changed to update multiple rows of **stock** as well.

The integrity risk is simply that the redundant copies of the data might not be accurate. If a lead time is changed in **manufact**, the **stock** column will be out of date until it, too, is updated. As with derived data, you should take pains to define the conditions under which the redundant data might be wrong.

Maximizing Concurrency

Some databases are used by only one program at a time; others are used concurrently by multiple programs. Two factors make concurrent programs inherently slower than doing the same amount of work serially.

- Multiple programs interfere with each other's use of buffers and disk drives. The pages read for one program might be driven from the buffers by the next program's query, and might have to be read again. Disk I/O for one query displaces the disk-access arm, slowing another program's sequential access.

- Programs that modify data lock pages, delaying all other programs that use the same data.

Easing Contention

Contention is inevitable between programs using the same resources. You can deal with it in three general ways:

1. Make programs use fewer resources, either by making them do less work, or by making them work more efficiently.

2. Arrange the resources better, for instance, by allocating tables to dbspaces to minimize contention.

3. Supply more resources: more memory, more and faster disk drives, more and faster computers.

The first point is the subject of Chapter 4, which deals with making queries faster. Sometimes you also must consider making queries do less, that is, reducing the functions available to on-line users. You can look for any allowed transaction that entails scanning an entire large table: any kind of summation, average, or management report, especially if it requires a join. Consider removing such transactions from on-line programs. Instead, offer a new facility that interactively schedules an off-peak job, with the output returned the next day.

Rescheduling Modifications

To the greatest extent possible, schedule modifications for times when no interactive users are using the database. One reason for this is that concurrent modifications must be done with all indexes in place; that means they incur all the time costs noted earlier. Another reason is that modifications not only lock table rows, they lock index pages as well. That increases the number of locks in use and the number of lock delays for all users.

The means of rescheduling modifications depend entirely on the details of your application.

Using an Update Journal

Instead of performing updates as the data becomes available, consider creating an *update journal*. This is a table that contains rows that represent pending updates. Its rows contain the following information:

- Data items to be changed in some base table, with null meaning no change
- Audit information that allows an erroneous update to be traced back to the transaction that entered the row

It is usually much faster to insert a new row in an update journal than actually to perform the update on one or more base tables. For one thing, the journal has at most one index to update, while the base tables usually have several. For another, locking delays are almost nil, since inserting a row in a table with no indexes requires only a lock on the inserted row. Another program can insert its row even before the first program commits its transaction.

Also, since no updates are performed on the primary tables, their pages are not locked. This allows query programs to run without delays.

After peak-usage hours (perhaps after close of business), you run a batch program to validate and apply the updates, row by row as they were inserted. You obviously would want to take pains to see that no update is forgotten or applied twice, regardless of possible system failures.

One way to update the base table is to use two cursors. The first cursor is a hold cursor; it is used to scan the rows of the journal. (See "Hold Cursors" on page 7-32.) For each row of the journal, the program goes through the following steps:

1. Issue the BEGIN WORK command.
2. Fetch the rows from the tables being updated using an update cursor. (This locks only these rows.)
3. Validate the update information from the journal against the data in the target rows.
4. Apply the updates to the target tables.
5. Update the journal row in some way to mark it as finished.
6. Issue the COMMIT WORK command (or the ROLLBACK WORK command, if an error is found).

You run a different program to drop and re-create the journal table only after every journal row is validated, applied, and marked.

The obvious disadvantage of an update journal is that the base tables do not reflect the most current data. If it is essential that updates be instantly visible, a journal scheme does not work.

The great advantages of reduced I/O and reduced locking delays during peak hours are a powerful argument in favor of an update journal. Deferred updates are accepted in many applications. For example, no bank promises to know your account balance more precisely than as of the close of business the preceding day, and for exactly this reason: the bank records transactions in a journal during the day and applies the updates overnight.

Isolating and Dispersing Updates

If updates really must be performed interactively during peak hours, you must find a different way to isolate queries from updates.

Splitting Tables to Isolate Volatile Columns

This idea was covered in an earlier section. If it is necessary to perform updates during peak hours, examine the structure of the tables. Can you separate the columns into volatile ones that are updated constantly, and static ones that are rarely updated? If so, consider splitting the tables so that the volatile columns are isolated in a companion table.

Depending on the mix of operations, and on the priorities of the users, you can then put either the static or the volatile subtable on your fastest disk drive.

Dispersing Bottleneck Tables

Small tables are sometimes used for summaries, usage records, and audits. For example, your interactive programs can maintain a table with a row for each authorized user. Each time the user starts or ends a program, it updates that user's row to show time on, time off, number of transactions, or other work-monitoring data.

Small tables that are only read do not cause performance problems, but small tables that are the subject of concurrent updates do. Any small table that must be updated on every transaction can become a bottleneck in which every on-line program queues up and waits.

To eliminate the problem, either use a journal with off-peak updates as described earlier, or disperse the bottleneck by creating many tables. To monitor users, create a one-row table for each user.

Summary

When tables are moderate in size and only one user at a time accesses the database, carefully applied relational theory is sufficient to produce good performance.

When both the number of tables and the number of users become larger and the response time begins to degrade, you must turn to practical solutions.

The first step is to understand and take advantage of the tools that **INFOR-MIX-OnLine** offers you. It permits you to arrange your tables on the hardware for maximum benefit. Then, one step at a time and always measuring, you can begin complicating the structure of the data model and the programs that use it.

11

Security, Stored Procedures, and Views

Chapter Overview

In some databases, all data is accessible to every user. In others, this is not the case; some users are denied access to some or all of the data. You can restrict access to data at these five levels:

1. When the database is stored in operating system files, you can sometimes use the file-permission features of the operating system.

    This level is not available when **INFORMIX-OnLine** holds the database. It manages its own disk space and the rules of the operating system do not apply.

2. You can use the GRANT and REVOKE statements to give or deny access to the database or to specific tables, and you can control the kinds of uses that people can make of the database.

3. You can use the CREATE PROCEDURE statement to write and compile a stored procedure that controls and monitors which users can read, modify, or create database tables.

4. You can use the CREATE VIEW statement to prepare a restricted or modified view of the data. The restriction can be vertical, excluding certain columns, or horizontal, excluding certain rows, or both.

5. You can combine GRANT and CREATE VIEW statements to achieve precise control over the parts of a table a user can modify and with what data.

These points are the subject of this chapter.

Controlling Access to Databases

The normal database-privilege mechanisms are based on the GRANT and REVOKE statements. They are covered in the section "Granting Privileges" on page 11-5. However, you can sometimes use the facilities of the operating system as an additional way to control access to a database.

Securing Database Files

Database servers other than **INFORMIX-OnLine** store databases in operating system files. Typically, a database is represented as a number of files: one for each table, one for the indexes on each table, and possibly others. The files are collected in a directory. The directory represents the database as a whole.

Multiuser Systems

You can deny access to the database by denying access to the database directory. The means by which you can do this depend on your operating system and your computer hardware. Multiuser operating systems have software facilities such as the Access Control List of VMS or the file permissions of UNIX.

*Note: In UNIX, the database directory is created with group identity **informix**, and the database server always runs under group identity **informix**. Thus, you cannot use group permissions to restrict access to a particular group of users. You can only remove all group permissions (file mode 700) and deny access to anyone except the owner of the directory.*

You also can deny access to individual tables in this way; for example, by making the files that represent those tables unavailable to certain users, while leaving the rest of the files accessible. However, the database servers are not designed with tricks of this kind in mind. When an unauthorized user tries to query one of the tables, the database server probably returns an error message about not being able to locate a file. This may confuse users.

Single-User Systems

Typical single-user systems have few software controls on file access; you can only make a database inaccessible to others by writing it on a disk that you can detach from the machine and keep locked.

None of these techniques apply when you use the **INFORMIX-OnLine** database server. It controls its own disk space at the device level, bypassing the file-access mechanisms of the operating system.

Securing Confidential Data

No matter what access controls the operating system gives you, when the contents of an entire database are highly sensitive you might not want to leave it on a public disk fixed to the machine. You can circumvent normal software controls when the data must be secure.

When the database is not being used by you or another authorized person, it does not have to be available on-line. You can make it inaccessible to everyone in several ways, with varying degrees of inconvenience:

• Detach the physical medium from the machine and take it away. If the disk itself is not removable, the disk drive might be removable.

• Copy the database directory to tape and take possession of the tape.

• Copy the database files using an encryption utility. Keep only the encrypted version.

In the latter two cases, after making the copies you must remember to erase the original database files using a program that overwrites an erased file with null data.

Instead of removing the entire database directory, you can copy and then erase the files that represent individual tables. However, do not overlook the fact that index files contain copies of the data from the indexed column or columns. Remove and erase the index files as well as the table files.

Granting Privileges

The authorization to use a database is called a *privilege*. For example, the authorization to use a database at all is called the Connect privilege, while the authorization to insert a row into a table is called the Insert privilege. You control the use of a database by granting these privileges to other users, or by revoking them.

Privileges are designed in two groups: one group affects the entire database and the other group relates to individual tables.

Database-Level Privileges

The three levels of database privilege provide an overall means of controlling who accesses a database.

Connect Privilege

The least of the privilege levels is Connect, which gives a user the basic ability to query and modify tables. Users with Connect privilege can perform the following functions:

- Execute the SELECT, INSERT, UPDATE, and DELETE statements, provided that they have the necessary table-level privileges.

- Execute a stored procedure, provided that they have the necessary table-level privileges.

- Create views, provided that they are permitted to query the tables on which the views are based.

- Create temporary tables and create indexes on the temporary tables.

Connect privilege is necessary before users can access a database at all. Ordinarily, in a database that does not contain highly sensitive or private data, you GRANT CONNECT TO PUBLIC shortly after creating the database.

The Users and the Public

Privileges are granted to single users by name, or to all users under the name of PUBLIC. Any grant to the public serves as a default privilege.

Prior to executing a statement, the database server determines whether the user has the necessary privileges. (The information is in the system catalog; see "Privileges in the System Catalog" on page 11-9.)

The database server looks first for privileges granted specifically to the requesting user. If it finds such a grant, it takes that information and stops. If there has been no grant to that user, the database server looks for privileges granted to public. If it finds a relevant one, it uses that.

Thus, you can set a minimum level of privilege for all users by granting privileges to public. You can override that in specific cases by granting higher individual privileges to users

---

If you do not grant Connect privilege to public, the only users who can access the database through the database server are those to whom you specifically grant the Connect privilege. If only certain users should have access, this is your means of providing it to them and denying it to all others.

Resource Privilege

The Resource privilege carries the same authorization as the Connect privilege. In addition, users with the Resource privilege can create new, permanent tables, indexes, and stored procedures, thus permanently allocating disk space.

Database Administrator Privilege

The highest level of database privilege is *Database Administrator*, or DBA. When you create a database, you are automatically the administrator. Holders of the DBA privilege can perform these functions:

- Execute the DROP DATABASE, START DATABASE, and ROLLFORWARD DATABASE commands.
- Alter the NEXT SIZE (but no other attribute) of the system catalog tables, and insert, delete, or update rows of any system catalog table except **systables**.

Warning: Although users with DBA privilege can modify the system catalog tables, it is strongly recommend that you do not update, delete, or alter any rows in the system catalog tables. Modifying the system catalog tables can destroy the integrity of the database.

- Drop or alter any object regardless of who owns it.
- Create tables, views, and indexes to be owned by other users.
- Grant database privileges, including DBA privilege, to another user.

Ownership Rights

The database, and every table, view, index, procedure, and synonym in it, has an owner. The owner of an object is usually the person who created it, although a user with DBA privilege can create objects to be owned by others.

The owner of an object has all rights to that object, and can alter or drop it without needing additional privileges.

Table-Level Privileges

You can apply seven privileges, table by table, to allow non-owners the privileges of owners. Four of them—the Select, Insert, Delete, and Update privileges—control access to the contents of the table. The Index privilege controls index creation. The Alter privilege controls the authorization to change the table definition. The References privilege controls the authorization to specify referential constraints on a table.

In an ANSI-compliant database, only the table owner has any privileges. In other databases, the database server, as part of creating a table, automatically grants all table privileges—except Alter and References—to public. This means that a newly created table is accessible to any user with Connect privilege. If this is not what you want—if there are users with Connect privilege who should not be able to access this table—you must revoke all privileges on the table from public after you create the table.

Access Privileges

Four privileges govern how users can access a table. As the owner of the table, you can grant or withhold these privileges independently:

- Select privilege allows selection, including selecting into temporary tables.
- Insert privilege allows a user to add new rows.
- Update privilege allows a user to modify existing rows.
- Delete privilege allows a user to delete rows.

Select privilege is necessary for a user to retrieve the contents of a table. However, Select privilege is not a precondition for the other privileges. A user can have Insert or Update privileges even when lacking Select privilege.

Privileges in the System Catalog

Privileges are recorded in the system catalog tables. Any user with Connect privilege can query the system catalog tables to determine what privileges have been granted and to whom.

Database privileges are recorded in the **sysusers** table, in which the primary key is user-id and the only other column contains a single character C, R, or D for the privilege level. A grant to the keyword of PUBLIC is reflected as a user name of **public** (lowercase).

Table-level privileges are recorded in **systabauth**, which uses a composite primary key of the table number, grantor, and grantee. In the **tabauth** column, the privileges are encoded in a six-letter list as follows:

A hyphen means an ungranted privilege, so that a grant of all privileges is shown as su-idxar while -u------ shows a grant of only Update. The code letters are normally lowercase, but they are uppercase when the keywords WITH GRANT OPTION are used in the GRANT statement.

When an asterisk appears in the third position, some column-level privilege exists for that table and grantee. The specific privilege is recorded in **syscolauth**. Its primary key is a composite of the table number, the grantor, the grantee, and the column number. The only attribute is a three-letter list showing the type of privilege: s, u, or r.

For example, your application might have a usage table. Every time a certain program is started, it inserts a row into the usage table to document that it was used. Before the program terminates, it updates that row to show how long it ran and perhaps to record counts of work performed by its user.

You want any user of the program to be able to insert and update rows in this usage table, so you grant Insert and Update privileges on it to public. However, you might grant Select privilege on it to only a few.

Index, Alter, and References Privileges

Index privilege permits its holder to create and alter indexes on the table. Index privilege, like Select, Insert, Update, and Delete privileges, is granted automatically to public when a table is created.

You can grant Index privilege to anyone, but to exercise the ability the user must also hold the Resource database privilege. So, although the Index privilege is granted automatically (except in ANSI-compliant databases), users who only have Connect privilege to the database are not able to exercise their Index privilege. This is reasonable since an index can fill a large amount of disk space.

Alter privilege permits its holder to use the ALTER TABLE statement on the table, including the power to add and drop columns, reset the starting point for SERIAL columns, and so on. You should grant Alter privilege only to users who understand the data model very well, and whom you trust to exercise their power very carefully.

The References privilege allows you to impose referential constraints on a table. As with Alter, you should only grant the References privilege to users who understand the data model very well.

Column-Level Privileges

You can qualify the Select, Update, and References privileges with the names of specific columns. This allows you to grant very specific access to a table: you can permit a user to see only certain columns, you can allow a user to update only certain columns, or you can allow a user to impose referential constraints on certain columns.

Using **INFORMIX-OnLine** (so that table data only can be inspected through a call to the database server), this feature solves the problem posed earlier: that only certain users should know the salary, performance review, or other sensitive attributes of an employee. To make the example specific, suppose there is a table of employee data defined as shown in Figure 11-1.

```
CREATE TABLE hr_data
    (
    emp_key INTEGER,
    emp_name CHAR(40),
    hire_date DATE,
    dept_num SMALLINT,
    user-id CHAR(18),
    salary DECIMAL(8,2)
    performance_level CHAR(1),
    performance_notes TEXT
    )
```

Figure 11-1   *A table of confidential employee information*

Since this table contains sensitive data, you execute the following statement immediately after creating it:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

For selected persons in the Human Resources department and for all managers, you execute a statement such as this one:

```
GRANT SELECT ON hr_data TO harold_r
```

In this way, you permit certain users to view all columns. (The final section of this chapter discusses a way to limit the view of managers to only their own employees.) For the first-line managers who carry out performance reviews, you could execute a statement like the following one:

```
GRANT UPDATE (performance_level, performance_notes)
    ON hr_data TO wallace_s, margot_t
```

This statement permits the managers to enter their evaluations of their employees. Only for the manager of the Human Resources department, or whoever is trusted to alter salary levels, would you execute a statement such as this one:

```
GRANT UPDATE (salary) ON hr_data to willard_b
```

For the clerks in the Human Resources department, you could execute a statement like this one:

```
GRANT UPDATE (emp_key, emp_name, hire_date, dept_num)
    ON hr_data TO marvin_t
```

This statement gives certain users the ability to maintain the nonsensitive columns but denies them authorization to change performance ratings or salaries. The person in the MIS department who assigns computer user-ids is the beneficiary of a statement such as this one:

```
GRANT UPDATE (user_id) ON hr_data TO eudora_b
```

On behalf of all users who are allowed to connect to the database but who are not authorized to see salaries or performance reviews, you execute statements such as this one to permit them to see the nonsensitive data:

```
GRANT SELECT (emp_key, emp_name, hire_date, dept_num, user-id)
    ON hr_data TO george_b, john_s
```

 Such users can perform queries such as this one:

```
SELECT COUNT(*) FROM hr_data WHERE dept_num IN (32,33,34)
```

However, any attempt to execute a query like the following one produces an error message and no data:

```
SELECT performance_level FROM hr_data
    WHERE emp_name LIKE "*Smythe"
```

Procedure-Level Privileges

You can apply the Execute privilege on a procedure to authorize non-owners to run a procedure. If you create a procedure in a database that is not ANSI-compliant, the default procedure-level privilege is Public; you do not need to grant Execute privilege to specific users unless you have first revoked it. If you create a procedure in an ANSI-compliant database, no other users have Execute privilege by default; you must grant specific users Execute privilege. The following example grants Execute privilege to the user **orion** so that **orion** can use the stored procedure named **read-address**:

```
GRANT EXECUTE ON read_address TO orion;
```

Procedure-level privileges are recorded in the **sysprocauth** system catalog table. The **sysprocauth** table uses a primary key of the procedure number, grantor, and grantee. In the **procauth** column, the execute privilege is indicated by a lowercase letter "e." If the execute privilege was granted with the WITH GRANT option, the privilege is represented by an uppercase letter "E."

Automating Privileges

It might seem that this design forces you to execute a tedious number of GRANT statements when you first set up the database. Furthermore, privileges require constant maintenance as people change their jobs. For example, if a clerk in Human Resources is terminated, you want to revoke the Update privilege as soon as possible; otherwise you risk the unhappy employee executing a statement like the following one:

```
UPDATE hr_data
    SET (emp_name, hire_date, dept_num) = (NULL, NULL, 0)
```

Less dramatic, but equally necessary, changes of privilege are required daily or even hourly, in any model that contains sensitive data. If you anticipate this need, you can prepare some automated tools to help in the maintenance of privileges.

Your first step should be to specify privilege classes that are based on the jobs of the users, not on the structure of the tables. For example, a first-line manager needs the following privileges:

- Select and limited Update privilege on the hypothetical **hr_data** table
- Connect privilege to this and other databases
- Some degree of privilege on several tables in those databases

When the manager is promoted to a staff position or sent to a field office, you must revoke all those privileges and grant a new set of privileges.

Define the privilege classes you support, and for each class specify the databases, tables, and columns to which you must give access. Then devise two automated procedures for each class: one to grant the class to a user, and one to revoke it.

Automating with INFORMIX-4GL

The mechanism you use depends on your operating system and other tools. If you are a programmer, the most flexible tool is probably **INFORMIX-4GL**. **4GL** makes it quite easy to program a simple user interaction, as in this fragment:

```
DEFINE mgr_id char(20)
PROMPT "What is the user-id of the new manager? " FOR mgr_id
CALL table_grant ("SELECT", "hr_data", mgr_id)
```

Unfortunately, although **INFORMIX-4GL** allows you to mix GRANT and REVOKE statements freely with other program statements, it does not permit you to create parameters from them from program variables. To customize a GRANT statement with a user-id taken from user input, the program must build the statement as a string, prepare it with a PREPARE statement, and execute it with an EXECUTE statement. (These statements are discussed in detail in Chapter 6, where the following example is analyzed in detail.)

Figure 11-2 shows one possible definition of the **4GL** function **table_grant()** that is invoked by the CALL statement in the preceding example.

```
FUNCTION table_grant (priv_to_grant, table_name, user_id)
    DEFINE  priv_to_grant char(100),{may include column-list}
            table_name CHAR(20),
            user_id CHAR(20),
            grant_stmt CHAR(200)
    LET grant_stmt =" GRANT ", priv_to_grant,
                    " ON ", table_name,
                    " TO ", user_id
    WHENEVER ERROR CONTINUE
    PREPARE the_grant FROM grant_stmt
    IF status = 0 THEN
        EXECUTE the_grant
    END IF
    IF status <> 0 THEN
        DISPLAY "Sorry, got error #", status, "attempting:"
        DISPLAY "    ", grant_stmt
    END IF
    WHENEVER ERROR STOP
END FUNCTION
```

Figure 11-2   A 4GL function that builds, prepares, and executes a GRANT statement

Automating with a Command Script

Your operating system probably supports automatic execution of command scripts. In most operating environments, **DB-Access** and **INFORMIX-SQL**, the interactive SQL tools, accept commands and SQL statements to execute from the command line. You can combine these two features to automate privilege maintenance.

The details depend on your operating system and the version of **DB-Access** or **INFORMIX-SQL** that you are using. In essence, you want to create a command script that performs the following functions:

- Takes as its parameter a user-id whose privileges are to be changed

- Prepares a file of GRANT or REVOKE statements customized to contain that user-id

- Invokes **DB-Access** or **INFORMIX-SQL** with parameters that tell it to select the database and execute the prepared file of GRANT or REVOKE statements

In this way, you can reduce the change of the privilege class of a user to one or two commands.

Using Stored Procedures

A stored procedure is a program written using the Informix Stored Procedure Language (SPL). Once you create a procedure, it is stored in an executable format in the database. It is a database object like a table, so anyone with appropriate privileges on the procedure can execute it. Stored procedures add value to the simple SQL code that they contain. Once written, they are simple to execute. They are also very efficient since they are stored in executable format.

You can use a stored procedure to control access to individual tables and columns in the database. You can accomplish various degrees of access control through a procedure. A powerful feature of SPL is the ability to designate a stored procedure as a DBA-privileged procedure. Writing a DBA-privileged procedure, you can allow users who have few or no table privileges to have DBA privileges when they execute the procedure. In the procedure, users can carry out very specific tasks with their temporary DBA privilege. The DBA-privileged feature allows you to accomplish the following tasks:

- You can restrict how much information individual users can read from a table.

- You can restrict all the changes made to the database and ensure that entire tables are not emptied or changed accidentally.

- You can monitor all of a certain class of changes made to a table, such as deletions or insertions.

- You can restrict all object creation (data definition) to occur within a stored procedure so that you have complete control over how tables, indexes, and views are built.

Creating and Executing Stored Procedures

To write a stored procedure, put the SQL statements that you want to be run as part of the procedure inside the CREATE PROCEDURE statement. You can also use the additional SPL statements to control the flow of operation within the statement. These additional statements include IF THEN ELSE, FOR, and others. (Stored Procedures and SPL are described fully in Chapter 8 of *The Informix Guide to SQL: Reference.*)

For example, if you want a user to be able to read the name and address of a customer, your stored procedure looks something like the one shown in Figure 11-3.

```
CREATE PROCEDURE read_address (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);

SELECT fname, lname, address, city, state, zipcode
    INTO p_fname, p_lname, p_city, p_state, p_zip
    FROM customer
    WHERE lname = lastname;

RETURN p_fname, p_lname, p_city, p_state, p_zip;

END PROCEDURE;
```

Figure 11-3   *Procedure to read from the* **customer** *table*

Stored procedures can simplify things for database users; they do not have to learn some of the complicated parts of SQL. Instead, the complicated activities can be simplified by someone who knows SQL—someone who writes a stored procedure to take care of an activity and lets others know that the procedure is stored in the database and that they can execute it. You can use procedures to your advantage by hiding the SQL functionality inside them and, in addition, controlling what happens to the database and the data in the database.

To execute, or run, a procedure, you use the EXECUTE PROCEDURE statement from **DB**-**Access** or from an embedded SQL program. To run the **read_address** procedure to see the full name and address of a customer named "Putnam," you use the following statement:

```
EXECUTE PROCEDURE read_address ("Putnum");
```

The following sections describe four ways to control access to data using a stored procedure.

Restricting Reads of Data

The procedure in Figure 11-3 hides the SQL syntax from users, but it requires that users have SELECT privilege on the **customer** table. If you want to restrict what users can select, you can write your procedure to work in the following environment:

- You are the DBA of the database.

- The users have CONNECT privilege to the database. They do not have SELECT privilege on the table.

- Your stored procedure (or set of stored procedures) is created using the DBA keyword.

- Your stored procedure (or set of stored procedures) reads from the table for users.

If you want users only to read the name, address, and telephone number of a customer, you can modify the procedure in Figure 11-3 as shown in Figure 11-4.

```
CREATE DBA PROCEDURE read_customer(cnum INT)
RETURNING CHAR(15), CHAR(15), CHAR(18);

DEFINE p_lname,p_fname CHAR(15);
DEFINE p_phone CHAR(18);

SELECT fname, lname, phone
    INTO p_fname, p_lname, p_phone
    FROM customer
    WHERE customer_num = cnum;

RETURN p_fname, p_lname, p_phone;

END PROCEDURE;
```

Figure 11-4   Stored procedure to restrict reads on customer data

Restricting Changes to Data

Using stored procedures, you can restrict changes made to a table. Simply channel all changes through a stored procedure. The stored procedure makes the changes, rather than users making the changes directly. If you want to

limit users to deleting one row at a time to ensure that they do not accidentally remove all the rows in the table, set up the database with the following privileges:

- You are the DBA of the database.

- All the users have Connect privilege to the database. They may or may not have Resource privilege. They do not have Delete (for this example) privilege on the table being protected.

- Your stored procedure is created using the DBA keyword.

- Your stored procedure performs the deletion.

Write a stored procedure similar to the one in Figure 11-5, which deletes rows from the **customer** table using a WHERE clause with the **customer_num** provided by the user.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DELETE FROM customer
    WHERE customer_num = cnum;

END PROCEDURE;
```

Figure 11-5   *Stored procedure to delete a row*

Monitoring Changes to Data

Using stored procedures, you can create a record of changes made to a database. You can record changes made by a particular user, or you can make a record of each time a change is made.

You can monitor all the changes made to the database by a single user. Simply channel all changes through stored procedures that keep track of changes made by each user. If you want to record each time the user **acctclrk** modifies the database, set up the database with the following privileges:

- You are the DBA of the database.

- All of the other users have Connect privilege to the database. They may or may not have Resource privilege. They do not have Delete (for this example) privilege on the table being protected.

- Your stored procedure is created using the DBA keyword.

- Your stored procedure performs the deletion and records that a change has been made by a certain user.

Write a stored procedure similar to the one in Figure 11-7, which updates a table using a customer number provided by the user. If the user happens to be **acctclrk**, a record of the deletion is put in the file "updates."

```
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" ;
ENF IF
END PROCEDURE;
```

Figure 11-6   ***Stored procedure to delete rows and record changes made by a certain user***

You can monitor all the deletions made through the procedure by removing the IF statement and making the SYSTEM statement more general. If you change the procedure in Figure 11-7 to record all deletions, it looks like the procedure in Figure 11-7.

```
CREATE DBA PROCEDURE delete_customer(cnum INT)

DEFINE username CHAR(8);
LET username = USER ;
DELETE FROM tbname WHERE customer_num = cnum;

SYSTEM
 "echo Deletion made from customer table, by "||username ||">>/hr/records/deletes";

END PROCEDURE;
```

Figure 11-7   ***Stored procedure to delete rows and record user***

Restricting Object Creation

To put restraints on what and how objects are built, use stored procedures within the following setting.

- You are the DBA of the database.

- All the other users have Connect privilege to the database. They do not have Resource privilege.

- Your stored procedure (or set of stored procedures) is created using the DBA keyword.

- Your stored procedure (or set of stored procedures) creates tables, indexes, and views however you defined them. You might use such a procedure to set up a training database environment.

Your procedure might include the creation of one or more tables and associated indexes, as shown in Figure 11-8.

```
CREATE DBA PROCEDURE all_objects()

CREATE TABLE learn1 (intone SERIAL, inttwo INT NOT NULL, charcol CHAR(10) )
CREATE INDEX learn_ix ON learn1 (inttwo).
CREATE TABLE toys (name CHAR(15) NOT NULL UNIQUE,
          description CHAR(30), on_hand INT);
END PROCEDURE;
```

Figure 11-8   ***DBA-mode procedure that adds tables and indexes to the database***

To use the **all_objects** procedure to control additions of columns to tables, revoke Resource privilege on the database from all users. When users try to create a table, index, or view using an SQL statement outside your procedure, they are not able to do so. When users execute the procedure, they have temporary DBA privilege so the CREATE TABLE statement, for example, succeeds, and you are guaranteed that every column added has a constraint placed on it. In addition, objects created by users are owned by that user. For the **all_objects** procedure, the two tables and the index are owned by whoever executed the procedure.

Using Views

A *view* is a synthetic table. You can query it as if it were a table, and in some cases you can update it as if it were a table. However, it is not a table, rather, it is a synthesis of the data that exists in real tables and other views.

The basis of a view is a SELECT statement. When you create a view, you define a SELECT statement that generates the contents of the view at the time the view is accessed. A user also queries a view using a SELECT statement. The database server merges the SELECT statement of the user with the one defined for the view and then actually performs the combined statements.

The result has the appearance of a table; it is enough like a table that a view even can be based on other views, or on joins of tables and other views.

Since you write a SELECT statement that determines the contents of the view, you can use views for any of these purposes:

- To restrict users to particular columns of tables

  You name only permitted columns in the select list in the view.

- To restrict users to particular rows of tables

  You specify a WHERE clause that returns only permitted rows.

- To constrain inserted and updated values to certain ranges

  You can use the WITH CHECK OPTION (discussed on page 11-27) to enforce constraints.

- To provide access to derived data without having to store redundant data in the database

  You write the expressions that derive the data into the select list in the view. Each time the view is queried, the data is derived anew. The derived data is always up to date, yet no redundancies are introduced into the data model.

- To hide the details of a complicated SELECT statement

  You hide complexities of a multitable join in the view so that neither users nor application programmers need to repeat them.

Creating Views

The following example creates a view based on a table in the demonstration database:

```
CREATE VIEW name_only AS
SELECT customer_num, fname, lname FROM customer
```

The view exposes only three columns of the table. Because it contains no WHERE clause, the view does not restrict the rows that can appear.

The following example is based on the join of two tables:

```
CREATE VIEW full_addr AS
SELECT address1, address2, city, state.sname, zipcode
    FROM customer, state
    WHERE customer.state = state.code
```

The table of state names reduces the redundancy of the database; it permits lengthy names such as Minnesota to be stored only once. This **full_addr** view lets users retrieve the address as if the full state name were stored in every row. The following two queries are equivalent:

```
SELECT * FROM full_addr WHERE customer_num = 105

SELECT address1, address2, city, state.sname, zipcode
    FROM customer, state
    WHERE customer.state = state.code
        AND customer_num = 105
```

However, you must use care when you define views that are based on joins. Such views are not *modifiable*; that is, you cannot use them with UPDATE, DELETE, or INSERT statements. (Modifying through views is covered beginning on page 11-25.)

The following example restricts the rows that can be seen in the view:

```
CREATE VIEW no_cal_cust AS
    SELECT * FROM customer WHERE NOT state = "CA"
```

This view exposes all columns of the **customer** table, but only certain rows. The next example is a view that restricts users to rows that are relevant to them:

```
CREATE VIEW my_calls AS
    SELECT * FROM cust_calls WHERE user_id = USER
```

All the columns of the **cust_calls** table are available, but only in those rows that contain the user-ids of the users who execute the query.

Duplicate Rows from Views

It is possible for a view to produce duplicate rows, even when the underlying table has only unique rows. If the view SELECT statement can return duplicate rows, then the view itself can appear to contain duplicate rows.

You can prevent this problem in two ways. One way is to specify DISTINCT in the select list in the view. However, that makes it impossible to modify through the view. The alternative is to always select a column or group of

columns that is constrained to be unique. (You can be sure only unique rows are returned if you select the columns of a primary key or of a candidate key. Primary and candidate keys are discussed in Chapter 8.)

Restrictions on Views

Since a view is not really a table, it cannot be indexed, and it cannot be the object of such statements as ALTER TABLE and RENAME TABLE. The columns of a view cannot be renamed with RENAME COLUMN. To change anything about the definition of a view, you must drop the view and re-create it.

Because it must be merged with the user's query, the SELECT statement on which a view is based cannot contain any of the following clauses:

INTO TEMP	The user's query might contain INTO TEMP; if the view contains it also, the data would not know where to go.
UNION	The user's query might contain UNION; indeed the query on the view might appear in a UNION clause in the user's query. No meaning has been defined for nested UNION clauses.
ORDER BY	The user's query might contain ORDER BY. If the view contains it also, the choice of columns or sort directions could be in conflict.

When the Basis Changes

The tables and views on which a view is based can change in several ways. The view automatically reflects most of the changes.

When a table or view is dropped, any views in the same database that depend on it are automatically dropped.

The only way to alter the definition of a view is to drop and re-create it. Therefore, if you change the definition of a view on which other views depend, you must also re-create the other views (since they all have been dropped).

When a table is renamed, any views in the same database that depend on it are modified to use the new name. When a column is renamed, views in the same database that depend on that table are updated to select the proper column. However, the names of columns in the views themselves are not changed. For an example of this, recall the following view on the **customer** table:

```
CREATE VIEW name_only AS
    SELECT customer_num, fname, lname FROM customer
```

Now suppose that the **customer** table is changed in the following way:

```
RENAME COLUMN customer.lname TO surname
```

To select last names of customers directly, you must now select the new column name. However, the name of the column as seen through the view is unchanged. The following two queries are equivalent:

```
SELECT fname, surname FROM customer

SELECT fname, lname FROM name_only
```

When you alter a table by dropping a column, views are not modified. If they are used, error -217 (Column not found in any table in the query) occurs. The reason views are not dropped is that you can change the order of columns in a table by dropping a column and then adding a new column of the same name. If you do this, views based on that table continue to work. They retain their original sequence of columns.

INFORMIX-**OnLine** permits you to base a view on tables and views in external databases. Changes to tables and views in other databases are not reflected in views. Such changes might not be apparent until someone queries the view and gets an error because an external table changed.

Modifying Through a View

It is possible to modify views as if they were tables. Some views can be modified and others not, depending on their SELECT statements. The restrictions are different depending on whether you use DELETE, UPDATE, or INSERT statements.

No modification is possible on a view when its SELECT statement contains any of the following features:

- A join of two or more tables

  Many anomalies arise if the database server tries to distribute modified data correctly across the joined tables.

- An aggregate function or the GROUP BY clause

  The rows of the view represent many combined rows of data; the database server cannot distribute modified data into them.

- The DISTINCT keyword or its synonym UNIQUE

  The rows of the view represent a selection from among possibly many duplicate rows; the database server cannot tell which of the original rows should receive the modification.

When a view avoids all these things, each row of the view corresponds to exactly one row of one table. Such a view is *modifiable*. (Of course, particular users can only modify a view if they have suitable privileges. Privileges on views are discussed beginning on page 11-29.)

Deleting Through a View

A modifiable view can be used with a DELETE statement as if it were a table. The database server deletes the proper row of the underlying table.

Updating a View

You can use a modifiable view with an UPDATE statement as if it were a table. However, a modifiable view can still contain derived columns, that is, columns that are produced by expressions in the select list of the CREATE VIEW statement. You cannot update derived columns (sometimes called *virtual* columns).

When a column is derived from a simple arithmetic combination of a column with a constant value (for example, order_date+30), the database server can, in principle, figure out how to invert the expression (in this case, by subtracting 30 from the update value) and perform the update. However, much more complicated expressions are possible, most of which cannot easily be inverted. Therefore, the database server does not support updating any derived column.

Figure 11-9 shows a modifiable view that contains a derived column, and an UPDATE statement that can be accepted against it.

```
CREATE VIEW call_response(user_id,received,resolved,duration)AS
    SELECT user_id,call_dtime,res_dtime,res_dtime-call_dtime
        FROM cust_calls
        WHERE user_id = USER

UPDATE call_response SET resolved = TODAY
    WHERE resolved IS NULL
```

Figure 11-9   A modifiable view and an UPDATE statement

The duration column of the view cannot be updated because it represents an expression (the database server cannot, even in principle, decide how to distribute an update value between the two columns named in the expression). But as long as no derived columns are named in the SET clause, the update can be performed as if the view were a table.

A view can return duplicate rows even though the rows of the underlying table are unique. You cannot distinguish one duplicate row from another. If you update one of a set of duplicate rows (for example, by using a cursor to update WHERE CURRENT), you cannot be sure which row in the underlying table receives the update.

Inserting into a View

You can insert rows into a view provided that the view is modifiable *and* contains no derived columns. The reason for the second restriction is that an inserted row must provide values for all columns, and the database server cannot tell how to distribute an inserted value through an expression. An attempt to insert into the **call_response** view shown in Figure 11-9 would fail.

When a modifiable view contains no derived columns, you can insert into it as if it were a table. However, the database server uses null as the value for any column that is not exposed by the view. If such a column does not allow nulls, an error occurs and the insert fails.

Using WITH CHECK OPTION

It is possible to insert into a view a row that does not satisfy the conditions of the view; that is, a row that is not visible through the view. It is also possible to update a row of a view so that it no longer satisfies the conditions of the view.

If this is improper, you can add the clause WITH CHECK OPTION when you create the view. This clause asks the database server to test every inserted or updated row to ensure that it meets the conditions set by the WHERE clause of the view. The database server rejects the operation with an error if the conditions are not met.

In Figure 11-9, the view named **call_response** was defined as follows:

```
CREATE VIEW call_response(user_id,received,resolved,duration)A
S
    SELECT user_id,call_dtime,res_dtime,res_dtime-call_dtime
        FROM cust_calls
        WHERE user_id = USER
```

It is possible to update the **user_id** column of the view, as in this example:

```
UPDATE call_response SET user_id = "lenora"
    WHERE received BETWEEN TODAY AND TODAY-7
```

The view requires rows in which **user_id** equals USER. If this update is performed by a user named **tony**, the updated rows vanish from the view. However, you can create the view as shown in this example:

```
CREATE VIEW call_response(user_id,received,resolved,duration)A
S
    SELECT user_id,call_dtime,res_dtime,res_dtime-call_dtime
        FROM cust_calls
        WHERE user_id = USER
WITH CHECK OPTION
```

Then the preceding update by **tony** is rejected as an error.

You can use the WITH CHECK OPTION feature to enforce any kind of data constraint that can be stated as a Boolean expression. For example, create a view of a table in which all the logical constraints on data are expressed as conditions of the WHERE clause. Then require all modifications to the table to be made through the view.

```
CREATE VIEW order_insert AS
    SELECT * FROM orders O
        WHERE order_date = TODAY -- no back-dated entries
            AND EXISTS -- ensure valid foreign key
                (SELECT * FROM customer C
                    WHERE O.customer_num = C.customer_num)
            AND ship_weight < 1000 -- reasonableness checks
            AND ship_charge < 1000
WITH CHECK OPTION
```

Because of EXISTS and other tests, all of which are expected to be successful when retrieving existing rows, this is a most inefficient view for displaying data from **orders**. However, if insertions to **orders** are made only through this view (and you are not already using integrity constraints to constrain data), it is impossible to insert a backdated order, an invalid customer number, or an excessive shipping weight and shipping charge.

Privileges and Views

When you *create* a view, the database server tests your privileges on the underlying tables and views. When you *use* a view, only your privileges with regard to the view itself are tested.

Privileges When Creating a View

When you create a view, the database server tests to make sure that you have all the privileges needed to execute the SELECT statement in the view definition. If you do not, the view is not created.

This test ensures that users cannot gain unauthorized access to a table by creating a view on the table and querying the view.

After you create the view, the database server grants you, the creator and owner of the view, at least Select privilege on it. No automatic grant is made to public as is the case with a newly created table.

The database server tests the view definition to see if the view is modifiable. If it is, the database server grants you Insert, Delete, and Update privileges on the view, provided that you also have those privileges on the underlying table or view. In other words, if the new view is modifiable, the database server copies your Insert, Delete, and Update privileges from the underlying table or view, and grants them on the new view. If you had only Insert privilege on the underlying table, you receive only Insert privilege on the view, and so on.

This test ensures that users cannot use a view to gain access to any privileges that they did not already have.

Since you cannot alter or index a view, the Alter and Index privileges are never granted on a view.

Privileges When Using a View

When you attempt to use a view, the database server only tests the privileges you have been granted on the view. It does *not* also test your right to access the underlying tables.

If you created the view, your privileges are the ones noted in the preceding paragraph. If you are not the creator, you have the privileges that were granted to you by the creator or someone who had privileges WITH GRANT OPTION.

The implication of this is that you can create a table and revoke public access to it. Then you can grant limited access privileges to the table through views. This can be demonstrated through the previous examples using the **hr_data** table. Its definition is repeated in Figure 11-10.

```
CREATE TABLE hr_data
    (
    emp_key INTEGER,
    emp_name CHAR(40),
    hire_date DATE,
    dept_num SMALLINT,
    user-id CHAR(18),
    salary DECIMAL(8,2)
    performance_level CHAR(1)
    performance_notes TEXT
    )
```

Figure 11-10   A table of confidential employee information (duplicate of Figure 11-1)

In Figure 11-10, the example centered on granting privileges directly on this table. The following examples take a different approach. Assume that when the table was created, the following statement was executed:

```
REVOKE ALL ON hr_data FROM PUBLIC
```

(This is not necessary in an ANSI-compliant database.) Now you create a series of views for different classes of users. For those who should have read-only access to the nonsensitive columns, you create the following view:

```
CREATE VIEW hr_public AS
    SELECT emp_key, emp_name, hire_date, dept_num, user_id
        FROM hr_data
```

Users who are given Select privilege for this view can see nonsensitive data and update nothing. For the clerks in the Human Resources department who must enter new rows, you create a different view:

```
CREATE VIEW hr_enter AS
    SELECT emp_key, emp_name, hire_date, dept_num
        FROM hr_data
```

You grant these users both Select and Insert privileges on this view. Because you, the creator of both the table and the view, have Insert privilege on the table and the view, you can grant Insert privilege on the view to others who have no privileges on the table.

On behalf of the clerk in the MIS department who enters or updates new user-ids, you create still another view:

```
CREATE VIEW hr_MIS AS
    SELECT emp_key, emp_name, user_id
        FROM hr_data
```

This view differs from the previous view in that it does not expose the department number and date of hire.

Finally, the managers need access to all columns and need the ability to update the performance-review data for their own employees only. These requirements can be met in the following way:

The table **hr_data** contains a department number and a computer user-id for each employee. Let it be a rule that the managers are members of the departments that they manage. Then the following view restricts managers to rows that reflect only their employees:

```
CREATE VIEW hr_mgr_data AS
    SELECT * FROM hr_data
        WHERE dept_num =
            (SELECT dept_num FROM hr_data
                WHERE user_id = USER)
        AND NOT user_id = USER
```

The final condition is required so that the managers do not have update access to their own row of the table. It is, therefore, safe to grant Update privilege to managers for this view, but only on selected columns, as in this statement:

```
GRANT SELECT, UPDATE (performance_level, performance_notes)
    ON hr_mgr_data TO peter_m
```

Summary

In a database that contains public material, or one that is used only by you and trusted associates, security is not an important consideration and few of the ideas in this chapter are needed. But as more and more people are allowed to use and modify the data, and as the data becomes more and more confidential, you must spend more time and be ever more ingenious at controlling the way people can approach the data.

The techniques discussed here can be divided into the following two groups:

- Keeping data confidential.

  When the database resides in operating system files you can use features of the operating system to deny access to the database. In any case, you control the granting of Connect privilege to keep people out of the database.

  When different classes of users have different degrees of authorization, you must allow them all Connect privilege. You can use table-level privileges to deny access to confidential tables or columns. Or, you can use a stored procedure to provide limited access to confidential tables or columns. In addition, you can deny all access to tables and allow it only through views that do not expose confidential rows or columns.

- Controlling changes to data and database structure.

  You safeguard the integrity of the data model by restricting grants of Resource, Alter, References, and DBA privileges. You ensure that only authorized persons modify the data by controlling the grants of Delete and Update privileges, and by granting Update privilege on as few columns as possible. You ensure that only consistent, reasonable data is entered by granting Insert privilege only on views that express logical constraints on the data. Alternatively, you can control the insertion and modification of data, or the modification of the database itself, by limiting access to constrictive stored procedures.

Networks and Distribution

12

Chapter Overview

It is easiest to think of the application, the database server, and the data as all being located in the same computer. But many other arrangements are possible. In some, the application and the database server are in different computers, and the data is distributed across several others.

This chapter discusses the possible arrangements at a conceptual level and points out some of their effects on performance and usability. It covers three kinds of networks:

- Local area networks (LANs)
- General networks in which applications work with database servers running in other (generally UNIX) machines.
- Distributed queries as implemented by **INFORMIX-STAR**.

To make a practical computer network work, you must master a multitude of technical details regarding hardware and software. There are far too many of these details, and they change too fast, to cover them in this book. Refer to the manual that accompanies the Informix client/server product that you use.

Network Configurations

Figure 12-1 shows a diagram of the simplest database configuration. The components of the system are as follows:

- An application program, which includes any program that issues a query. Besides programs written in **INFORMIX-4GL** or another language with embedded SQL, it includes compiled screen forms and reports.
- A database server that receives SQL statements from the application, and returns selected rows of data to it.
- An operating system that manages the computer file system. Some database servers manage disk space directly.
- A disk device where the tables are actually stored
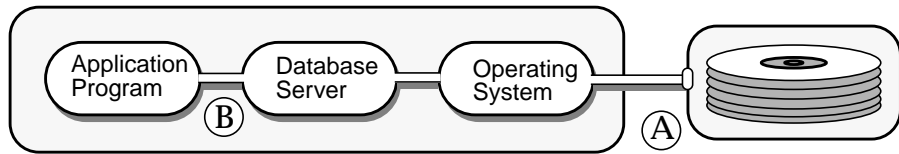
Figure 12-1   ***A simple database configuration***

Figure 12-1 describes a wide range of systems. For example, the operating system could be PC-DOS or MS-DOS, the database server could be **INFORMIX**-**SE**, and the program could be written in **INFORMIX**-**4GL**.

Or, with one minor change, Figure 12-1 could depict either the UNIX or the NetWare operating system and the **INFORMIX**-**OnLine** database server. The change is the depiction of not one but several application programs running concurrently, all receiving service from a corresponding database server.

The common theme in these configurations is that all the components run in the same computer. However, you can divide the system at either of the points marked *A* and *B* in the diagram, insert a layer of software and hardware, and thereby distribute the system to two or more computers.

The Local Area Network

One configuration, in which the division occurs at point *A* in Figure 12-1, is the local area network (LAN). In a LAN, users have a computer that contains an operating system, a database server, and an application. However, the disks that contain the tables are attached to other computers called *file servers*, as shown in Figure 12-2.

The file servers dispense disk files on request to the other machines. The LAN software makes it seem, for all practical purposes, that the disks of the file server are attached directly to each user's computer. The Informix database engine for PC-DOS and MS-DOS supports this configuration.

The advantage of a LAN is that the database tables are held in a single place but are still accessible to all users on the network. A LAN can be extended to serve more users easily and economically.
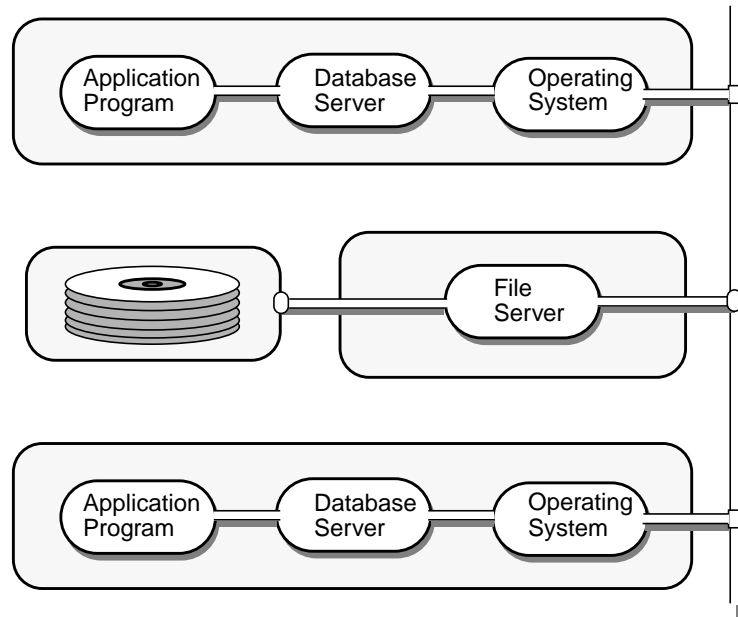
Figure 12-2   Configuration of a LAN

The LAN configuration is at a disadvantage in performance. Any data that the database server reads must pass over the network from a file server. This procedure is always slower than access to a local disk. Moreover, the requests for disk I/O from all users must pass over the network and through the file server, which can lead to delays when several users are busy at once.

Networking the Database Server

An obvious answer to the problems of the LAN is to split Figure 12-1 at the point marked *B*; that is, to move the database server away from the application and into the computer to which the disks are attached. That way, only the processed data—the selected rows that satisfy the query—pass over the network.

The **INFORMIX-NET** products support this configuration. There are several **INFORMIX-NET** products, each one for a different type of network. A typical configuration is depicted in Figure 12-3.

A network can include one or more computers that contain database servers. Figure 12-3 shows one computer. It contains a database server that might be **INFORMIX-SE**, **INFORMIX-OnLine**, or **INFORMIX-OnLine for NetWare**. It is possible to have different database servers on a network in different machines. It is also possible to have more than one copy of each database server on the network, in the same machine, or in different machines.

Each database server is a network service. The way a network service is defined and made available to the network depends on the type of client/server software in use.
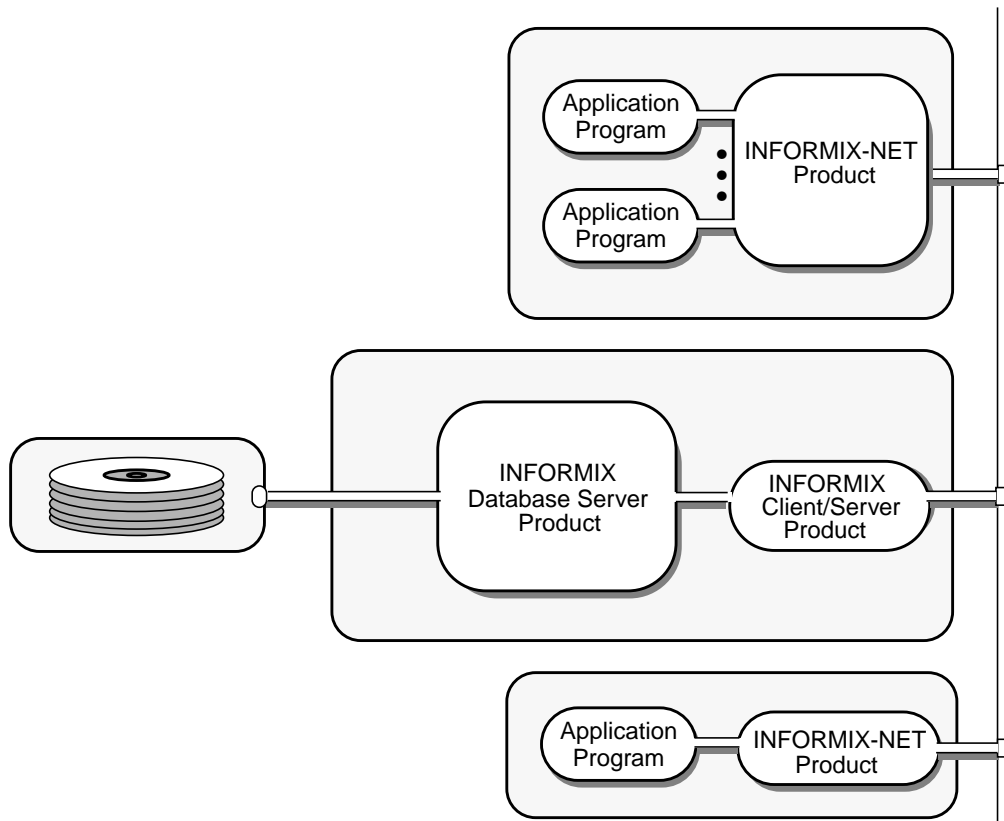
Figure 12-3   Typical configuration of **INFORMIX-NET** *products*

Figure 12-3 depicts two machines that contain application programs. The upper one contains several applications. This machine, since it supports multiple concurrent programs, must be running an operating system such as UNIX.

The lower machine in Figure 12-3 contains only one application. It can be running PC-DOS or MS-DOS. All of these configurations are supported by a version of **INFORMIX-NET**.

Network Transparency

It is important to realize that in all of these network configurations the application programs are basically unchanged. There is nothing special about a report, form, or program that works with a network database.

Except for one possible change discussed in the next section (the use of database server names when referring to databases), it is possible to run an application against a local database and later run the identical application against a database in another machine. The program runs equally well in either case. In fact, as long as the data model is the same, a program cannot tell the difference between a local database server and a distant one.

Connecting to Data

When the application and its data are moved to separate computers, two questions immediately arise: How can they be connected again? How does the application find its data that is now somewhere else?

Connecting in the LAN

The LAN is designed to make the disks attached to file servers appear to be attached to the user's machine. As a result, databases and tables are found just as they normally are, by specifying the disk and directory where they are kept.

The database server uses an environment variable named DBPATH to find a database. If the path to a database specifies a disk that is elsewhere on the network, the database server uses it just the same and never knows the difference.

Informix DOS engines allow you to create tables in specific locations, as in the following example:

```
CREATE TABLE net_tab
    (
    col_1 integer,
    col_2 float
    )
    IN "F:\tables"
```

This feature gives you the ability to place a table on a specific disk. You can, for instance, put different tables on different file servers to balance network load.

Note: The Informix UNIX database servers allow you to create a table in a particular pathname (INFORMIX-SE) or dbspace (INFORMIX-OnLine), but all tables must be on disks attached to the computer in which the database server is running.

Connecting Through INFORMIX-NET

INFORMIX-NET products support a variety of network types, operating systems, and database servers. While the basic method for connecting an application to its data is constant, there are special cases for every combination of operating system, network, and database server. Only the basic ideas are covered here. Consult the document for the **INFORMIX-NET** product you are using for details.

The Role of INFORMIX-NET

When an application begins execution, it uses an environment variable to find the database server to use. Before **INFORMIX-NET** is installed, the application finds a database server. Afterward, it finds **INFORMIX-NET** instead. **INFORMIX-NET** monitors the requests of the application and passes them to the actual database server. The application is not aware of this extra layer of software.

Opening a Database

Before it can do anything else, an application must *open* a database. This is most often done by executing the DATABASE statement. More rarely, it is done with the CREATE DATABASE statement, which creates a database and then opens it. Only a few other SQL statements, such as DROP DATABASE, can be executed until a database is opened.

At the moment of opening a database, **INFORMIX-NET** determines the location of the database and finds the database server that controls it. From information in the DATABASE or CREATE DATABASE statement, and from the contents of the DBPATH environment variable, **INFORMIX-NET** deduces whether the database is located in the same machine as the application, or in a different machine. If it is in a different machine, **INFORMIX-NET** establishes communication with a database server in that machine, and then acts as a conduit to pass requests and data between the application and the database server.

Explicit Locations Using File Specifications

You can write the location of a database as part of the database name. The following example might be written in an application running on a DOS machine:

```
DATABASE "D:\DBS\TESTDB"
```

Here is a similar example from a UNIX-based application. (You can tell because the slashes lean the other way.)

```
DATABASE "/usr/maxtel/testdb"
```

The database is probably local. **INFORMIX-NET** starts a copy of the **INFORMIX-SE** database server in this machine, and the application uses that database server.

On some UNIX systems, a DATABASE statement with the identical form can point to a different machine, as follows:

```
DATABASE "/subtle/dbdir/testdb"
```

In this example, you must suppose that the UNIX file system named **subtle** is physically attached to a different machine. It is mounted to the application machine using the Network File System (NFS) feature or the Remote File Sharing (RFS) feature. Nothing in the file specification reflects that fact. However, **INFORMIX-NET** recognizes that the file system is not local to the application machine. It starts a copy of **INFORMIX-SE** in the machine that owns **subtle** and passes the requests of the application to it.

Two slashes at the start of a database file specification introduce the name of a network machine. They are always forward slashes, even when used in a DOS application. Here is an example:

```
DATABASE "//avignon/usr/maxtel/testdb"
```

The double slash at the start of the pathname tells **INFORMIX-NET** that the database is located on a different machine—in this case, the machine known to the network as **avignon**. Since a complete file specification follows, **INFORMIX-SE** must manage the database. (**INFORMIX-OnLine** does not use the normal file system.) **INFORMIX-NET** starts a database server in machine **avignon** and acts as a conduit between it and the application.

Explicit Location by Database Server Name

When the complete file specification is not required, you can specify the location of the database in a different form, as in this example:

```
DATABASE telefonen@munchen
```

The database name is **telefonen**; it is located in the machine known to the network as **munchen**. You use this method to specify the location of a database when the database is managed by the **INFORMIX-OnLine** database server.

You can use the same form of the statement with the CREATE DATABASE and DROP DATABASE commands. However, when you write an explicit location into one of these commands, you limit the application to that location. It must be changed to use a database at a different location.

Implicit Database Locations

Quite often the database name in the command consists of nothing but an identifier, giving no clue as to location. This is the best way to write the DATABASE, CREATE DATABASE, and DROP DATABASE statements because you do not need to change the application when the database location changes.

When the application tries to open a database giving only a name, **INFORMIX-NET** looks at the DBPATH environment variable for information.

This environment variable lists all the locations where a database might be found. When a network is not used, the contents of the variable tell **INFORMIX-SE** the file directories where a database might be found.

If no DBPATH variable is defined, **INFORMIX**-**NET** takes a default action. If **INFORMIX**-**OnLine** is installed on this machine, it is used. Otherwise the database is managed by **INFORMIX**-**SE** and is located in the current directory.

When DBPATH exists, **INFORMIX**-**NET** consults the contents of the variable to find where the database is located. Three kinds of items can be listed in DBPATH:

- A local directory

  If the named database appears there, the local **INFORMIX**-**SE** is used to access it.

- In a UNIX system, the path to a directory mounted by NFS or RFS

  If the named database appears there, a copy of **INFORMIX**-**SE** is started on the machine that owns the file system, and it is used for access.

- The name of a remote server, preceded by a double slash

  INFORMIX-**NET** queries that network site to find out which kind of database server it is, and whether it can open a database of the specified name.

In this way, it is possible to take an application written for a local database and, by changing the environment variable DBPATH, make it work with a database located in a different computer.

Distributed Data

While a LAN or other network allows you to separate the application from the data, the application still is limited to the contents of a single database. With most database servers, you only can query or modify tables in the current database.

The **INFORMIX**-**OnLine** database server allows you to query data in any database that it controls. That is, you can query and modify tables in databases other than the current database, as long as those other databases are managed by the same **INFORMIX**-**OnLine** database server (and as long as **INFORMIX**-**STAR** also resides on the same machine as the **OnLine** database server).

When the **INFORMIX**-**STAR** product is added to **INFORMIX**-**OnLine**, it becomes possible to query and modify tables in databases that are managed by any **INFORMIX**-**OnLine** database server anywhere in your network.

When the **INFORMIX**-**TP/XA** feature is added to **INFORMIX**-**OnLine** (with **INFORMIX**-**ESQL/C**), you can create global transactions that span multiple computer systems and even multiple XA-compliant database systems from different vendors.

Naming External Tables

The database that is opened by the DATABASE or CREATE DATABASE statement is the *current* database. Any other database is an *external* database. To refer to a table in an external database, you include the database name as part of the table name.

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

The external database is **salesdb**. The table in it is named **contacts**. You can use the same notation in a join. When there is more than one external table, the long table names can become cumbersome unless you use aliases to shorten them.

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

You can qualify a database name with a *server name*, the name of a network machine where another **INFORMIX-OnLine** database server is running. That is how you specify an external database that is managed by a different database server.

```
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 external tables are being joined. The joined rows are being stored in a temporary table in the current database. The external tables are located in two different database servers. One is named **central**, the other **boston**.

It is always permissible to *overspecify* a table name or database name. That is, even if the current database is **masterdb** and the current database server is **central**, it is permissible in a query to refer to a table named **masterdb@central:customer**.

Using Synonyms with External Tables

A *synonym* is a name that you can use in place of another name. The main use of synonyms is to make it more convenient to refer to external tables. You can also use synonyms to disguise the location of external tables.

Here is the preceding example, revised to use synonyms for the external tables:

```
CREATE SYNONYM mcust FOR masterdb@central:customer;

CREATE SYNONYM bords FOR sales@boston:orders;

SELECT bords.order_num, mcust.fname, mcust.lname
    FROM mcust, bords
    WHERE mcust.customer_num = bords.Customer_num
    INTO TEMP mycopy
```

The CREATE SYNONYM statement stores the synonym name in the system catalog table **syssyntable** in the current database. The synonym is available to any query made within that database.

A short synonym makes it easier to write queries, but synonyms can play another role. They allow you to move a table to a different database, or even to a different computer, while leaving your queries the same.

Suppose you have a number of queries that refer to the table names **customer** and **orders**. The queries are embedded in programs, forms, and reports. The tables are part of database **stores5**, which is kept on database server **avignon**.

Now the decision is made that the same programs, forms, and reports are to be made available to users of a different computer on the network (database server **nantes**). Those users have a database that contains a table named **orders** containing the orders at their location, but they need access to the table **customer** at **avignon** (or else the table has to be duplicated, which creates problems of maintenance).

To those users, the **customer** table is external. Does this mean you must prepare special versions of the programs and reports, versions in which the **customer** table is qualified with a database server name? A better solution is to create a synonym in the users' database:

```
DATABASE stores5@nantes;
CREATE SYNONYM customer FOR stores5@avignon:customer
```

When the stored queries are executed in your database, the name **customer** refers to the actual table. When they are executed in the other database, the name is translated through the synonym into a reference to the external table.

Synonym Chains

To continue the preceding example, suppose that a new computer is added to your network. Its name is **db_crunch**. The **customer** table and other tables are moved to it to reduce the load on **avignon**. You can reproduce the table on the new database server easily enough, but how can you redirect all accesses to it? One way is to install a synonym to replace the old table.

```
DATABASE stores5@avignon EXCLUSIVE;
RENAME TABLE customer TO old_cust;
CREATE SYNONYM customer FOR stores5@db_crunch:customer;
CLOSE DATABASE
```

When you execute a query within **stores5@avignon**, a reference to table **customer** finds the synonym and is redirected to the version on the new computer. This is also true of queries executed from database server **nantes**. The synonym in the database **stores5@nantes** still redirects references to **customer** to database **stores5@avignon**; however, the new synonym there sends the query to database **stores5@db_crunch**.

Chains of synonyms can be useful when, as in this example, you want to redirect all access to a table in one operation. However, as soon as possible you should update all users' databases so their synonyms point directly to the table. There is extra overhead in handling the extra synonyms and, if any computer in the chain is down, the table cannot be found.

Modifying External Tables

The following example shows how you can modify external tables:

```
DATABASE stores5@nantes;
BEGIN WORK;
UPDATE stores5@avignon:customer
    SET phone = "767-8592"
    WHERE customer_num = 149;
COMMIT WORK
```

You can modify more than one table in the same transaction. However, the same **INFORMIX-OnLine** database server (if you are not using **OnLine** with **INFORMIX-STAR** or **INFORMIX-TP/XA**) must manage all the tables in a single transaction. In the preceding example, an external table at database server **avignon** is updated. You also can modify other tables at **avignon** in the same transaction. However, you cannot modify a table in the current database, since the current database is managed by **nantes**.

Summary

The simplest network, the LAN, lets multiple workstations share a disk. Each workstation has its application program and its own copy of the database server, but the tables it uses can be located on disks attached to one or more computers. Multiple workstations share the data, but all input and output travels over the network and that can cause a performance problem.

The more general form of network supported by **INFORMIX**-**NET** (and **INFORMIX**-**STAR**) products separates the application from the database server. The application runs in one machine, while the database server operates in the computer to which the data is physically attached. There are many possible combinations of network software, operating systems, and database servers, and each has its subtleties that must be mastered.

In general networking, the crucial moment occurs when the application opens a database. At that moment, based on information in the command or in an environment variable, the network software locates the database and sets up communication to the database server that manages it.

When you use **INFORMIX**-**OnLine** with the **INFORMIX**-**STAR** feature installed, you can query and modify tables from multiple databases managed by multiple **INFORMIX**-**OnLine** database servers. You can use synonyms to disguise the locations of these tables.

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

> IBM Director of Licensing
> IBM Corporation
> North Castle Drive
> Armonk, NY 10504-1785
> U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

> IBM World Trade Asia Corporation
> Licensing
> 2-31 Roppongi 3-chome, Minato-ku
> Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

> IBM Corporation
> J74/G4
> 555 Bailey Ave
> P.O. Box 49023
> San Jose, CA 95161-9023
> U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Index

This index covers both the Tutorial and the Reference Manual.
Page numbers that end in T can be found in the Tutorial, and those
that end in R can be found in the Reference Manual.

in relational expressions 2-24T
multiplying or dividing values 3-25R
precision and size 9-14T
range of expressions 3-21R
range of operations with DATE and
DATETIME 3-20R
syntax 7-351R, 7-404R
with EXTEND function 3-21R, 3-23R
See also Literal INTERVAL.
INTERVAL Field Qualifier segment
7-397R
INTO keyword
choice of location 6-21T
in FETCH statement 6-21T
in SELECT 7-253R
mismatch signalled in SQLAWARN
6-12T
restrictions in INSERT 5-10T
restrictions in prepared statement
6-28T
retrieving multiple rows 6-19T
retrieving single rows 6-13T
syntax
in FETCH 7-144R
in SELECT 7-246R
use
in FETCH 7-148R
in SELECT 7-253R
INTO TEMP keywords
description of 2-70T
restrictions in view 11-24T
syntax in SELECT 7-246R
use
in SELECT 7-274R
with UNION operator 7-276R
IS keyword
in Condition segment 7-337R
with WHERE keyword 7-262R
IS NOT keywords, syntax in Condition
segment 7-332R
IS NULL keywords 7-262R
ISAM error code 6-12T
Isolation level
Committed Read 7-25T, 7-295R
Cursor Stability 7-25T, 7-295R
definitions 7-295R
Dirty Read 7-24T, 7-295R
in external tables 7-296R, 7-298R
Repeatable Read 7-26T, 7-295R
setting 7-24T

items table in stores5 database, columns
in 1-5R
ITYPE field
description of 6-8R, 6-15R
setting with SET DESCRIPTOR
7-286R
with X/Open programs 7-161R

J

Join
associative 2-64T
creating 2-59T
definition of 2-8T
effect of large join on optimization
4-14T
equi-join 2-59T
in Condition segment 7-266R
multiple-table join 2-64T, 7-267R
natural 2-62T
outer join 3-19T, 7-268R
restrictions in modifiable view 11-25T
self-join 3-11T, 7-267R
sort merge 4-13T
two-table join 7-266R
Join column. *See* Foreign key.
Join condition. *See* Condition segment.
Journal updates 10-33T

K

Key lock 7-22T
Key, candidate. *See* Candidate key.
Key, foreign. *See* Foreign key.
Key, primary. *See* Primary key
constraint.

L

Label 2-41T, 3-42T
LAN. *See* Local Area Network.
LAST keyword
syntax in FETCH 7-144R
use in FETCH 7-146R
Latency 4-23T
LENGTH field
description of 6-7R, 6-15R
setting with SET DESCRIPTOR
7-285R

Model. *See* Data model.

MODIFY keyword
    syntax in ALTER TABLE 7-12R
    use in ALTER TABLE 7-23R

MODIFY NEXT SIZE keywords
    syntax in ALTER TABLE 7-12R
    use in ALTER TABLE 7-29R

MONEY data type 9-12T
    changing data types 3-18R
    description of 9-11T, 3-14R
    display format 9-12T
    in INSERT 5-8T
    syntax 7-351R
    using as default value 7-16R, 7-74R
    See also DECIMAL data type.

MONEY value
    setting DBMONEY environment
        variable 4-10R
    specifying European format with
        DBMONEY 4-11R

MONTH function
    as time function 2-46T
    syntax in expression 7-366R
    use in expression 7-368R

MONTH keyword
    syntax
        in DATETIME data type 7-353R
        in INTERVAL data type 7-398R
    use
        as DATETIME field qualifier 3-7R,
            7-401R
        as INTERVAL field qualifier
            3-12R, 7-404R

Multi-row query
    destination of returned values 7-147R
    managing with FETCH 7-145R

N

NAME field, description of 6-7R, 6-15R

Naming convention
    column 7-73R, 7-231R
    database 7-347R
    index 7-346R, 7-396R, 7-415R
    table 7-68R, 7-382R, 7-417R
    See also Database Name segment.
    See also Index Name segment.
    See also Table Name segment.

Natural join 2-62T

Nested ordering, in SELECT 7-273R

Network
    data sent over 4-26T
    performance of 4-25T
    simple model of 4-26T

Network environment variable
    DBNETTYPE 4-11R
    DBPATH 4-11R
    SQLRM 4-21R
    SQLRMDIR 4-22R

Network File System (NFS) 12-9T

Networking
    configurations 12-3T
    connecting to data 12-7T
    description of 12-3T
    distributed data 12-11T
    file server 12-4T
    Local Area Network (LAN) 12-4T

NEXT keyword
    syntax in FETCH 7-144R
    use in FETCH 7-146R

NEXT SIZE keywords
    specifying size of extents 10-8T
    use in CREATE TABLE 7-89R
    use in GRANT 7-168R

NFS. *See* Network File System.

Nonsequential access. *See* Disk access,
    nonsequential.

NOSORTINDEX environment variable
    4-20R

NOT CLUSTER keywords
    syntax in ALTER INDEX 7-10R
    use in ALTER TABLE 7-11R

NOT FOUND keywords, in the
    WHENEVER statement 7-323R,
    7-326R

NOT IN keywords, use in Condition
    subquery 7-341R

NOT keyword
    syntax
        in Condition segment 7-331R,
            7-332R
        with BETWEEN keyword 7-261R
        with IN keyword 7-263R
    use
        in Condition segment 7-338R
        with LIKE, MATCHES keywords
            7-262R

NOT logical operator 2-30T

NOT NULL keywords
  syntax
    in ALTER TABLE 7-14R
    in CREATE TABLE 7-73R
  use
    in ALTER TABLE 7-23R
    in CREATE TABLE 9-26T, 7-75R
    with IS keyword 7-262R

NOT relational operator 2-27T

NOT WAIT keywords, in SET LOCK
    MODE 7-297R

Not-equals relational operator 2-26T

NOTFOUND keyword, contrasted with
    NOT FOUND keywords 7-326R

NULL keyword, ambiguous as
    procedure variable 7-392R

NULL relational operator 2-30T

Null value
  checking for in SELECT 7-253R
  detecting in ESQL 6-15T
  in IF statement 8-56R
  in INSERT statement 5-7T
  restrictions in primary key 8-7T
  restrictions on primary key 7-79R
  returned implicitly by stored
    procedure 8-69R
  specifying as default value 7-17R
  testing for 2-30T
  with logical operator 2-30T
  with WHILE statement 8-77R

NULLABLE field, description of 6-7R,
    6-15R

NUMERIC data type. *See* DECIMAL
    data type.

O

OF keyword
  syntax in DECLARE 7-102R
  use in DECLARE 7-110R

ON EXCEPTION statement
  placement of 8-63R
  scope of control 8-31R
  syntax 8-62R
  trapping errors 8-29R
  user-generated errors 8-31R

ON keyword
  syntax
    in CREATE INDEX 7-48R

    in GRANT 7-165R
    in REVOKE 7-236R
  use
    in CREATE INDEX 7-49R
    in GRANT 7-171R

On-line
  files 6T, 10R
  help 6T, 10R

OPEN statement
  activating a cursor 6-19T
  constructing the active set 7-198R
  opening an insert cursor 7-199R
  opening select or update cursors
    6-19T, 7-198R
  reopening a cursor 7-199R, 7-200R
  substituting values for ? parameters
    7-200R
  syntax 7-197R
  when to use FOREACH 7-203R
  with concatenation operator 7-357R
  with FREE 7-204R

Opening a cursor 6-19T, 6-23T

Optimization, specifying a high or low
    level 7-301R

Optimizer
  and GROUP BY 4-13T, 4-17T, 4-18T
  and ORDER BY 4-13T, 4-16T, 4-18T
  and SET OPTIMIZATION statement
    4-14T, 7-301R
  autoindex path 4-32T
  composite index use 4-32T
  description of 4-8T
  disk access 4-21T
  display query plan 4-18T
  filter selectivity 4-15T
  index not used by 4-31T, 4-33T
  index used by 4-14T, 4-15T
  methods of 4-13T
  query plan 4-8T
  sort merge join 4-13T
  sorting 4-18T
  specifying high or low level of
    optimization 4-14T
  system catalog use 4-9T, 4-14T
  when index not used 4-34T

Optimizing
  a query 7-288R
  a server 7-301R
  across a network 7-302R
  techniques 4-3T
  updating system catalog tables 7-321R

OR keyword
  syntax in Condition segment 7-331R
  use in Condition segment 7-344R
OR logical operator 2-30T
OR relational operator 2-28T
ORDER BY keywords
  ascending order 2-13T, 7-272R
  DESC keyword 2-13T, 2-22T
  descending order 7-272R
  display label with 2-43T
  indexes for 4-16T, 4-30T
  multiple columns 2-13T
  relation to GROUP BY 3-7T
  restrictions in INSERT 5-10T, 7-187R
  restrictions in view 11-24T
  restrictions with FOR UPDATE 7-8T
  select columns by number 2-21T,
      7-273R
  sorting rows 2-12T, 4-18T
  syntax in SELECT 7-246R
  use
      in SELECT 7-271R
      with UNION operator 7-275R
orders table in stores5 database, columns
      in 1-5R
Outer join
  description of 3-19T
  nested 3-25T
OUTER keyword, with FROM keyword
      in SELECT 7-258R
OUTPUT statement, syntax and use
      7-206R
Owner
  in ALTER TABLE 7-13R
  in CREATE SYNONYM 7-63R
  in Index Name segment 7-346R,
      7-396R, 7-407R, 7-414R
  in RENAME COLUMN 7-231R
  in RENAME TABLE 7-233R
  in Table Name segment 7-382R,
      7-417R
  in View Name segment 7-420R
  of view in CREATE VIEW 7-420R
Owner-privileged procedure 8-15R
Ownership 11-7T

P

Page 10-4T
Page buffer
  cost of nonsequential access 4-24T
  description of 4-22T
  effect on performance 4-23T
  restrictions with BLOB data 10-19T
PAGE keyword
  use in ALTER TABLE 7-30R
  use in CREATE TABLE 7-90R
Page lock 7-22T
Parameter
  BYTE or TEXT in SPL 8-45R
  in CALL statement 8-35R
  to a stored procedure 8-27R
Parameterizing a statement
  in PREPARE 7-213R
  with SQL identifiers 7-214R
Parent-child relationship 7-19R, 7-80R
Parts explosion 6-25T
PATH environment variable 4-23R
Pathname
  including in SQLEXEC 4-21R
  specifying with DBPATH 4-12R
  specifying with DBSRC 4-14R
  specifying with PATH 4-23R
Percent (%) sign, wildcard in Condition
      segment 7-338R
PERFORM keyword, in the WHENEVER
      statement 7-323R
Performance
  adding indexes 4-30T
  assigning table to dedicated disk
      10-6T
  bottleneck tables 10-34T
  buffered log 9-24T
  clustered index 10-24T
  defragmenting a dbspace 10-10T
  depends on concurrency 7-17T
  disk access 4-21T, 4-23T, 4-30T
  disk access by rowid 4-24T
  disk arm motion 10-8T
  disk latency 4-23T
  dropping indexes to speed
      modifications 10-23T
  duplicate keys slow modifications
      10-22T
  effect of BLOB location 10-18T
  effect of correlated subquery 4-33T

effect of filter expression 4-10T, 4-33T,
    4-34T
effect of index damage 4-32T
effect of indexes 4-12T, 10-21T to
    10-22T
effect of locking 7-18T
effect of optimizer 4-8T
effect of regular expressions 4-33T
effect of table size 4-25T, 4-30T
effect of updates 4-32T
filter selectivity 4-15T
fragmentation 10-10T
"hot spots," finding 4-7T
how indexes affect 4-24T
improved by specifying optimization
    level 4-14T
improved with temporary table 4-34T
improving 4-27T
increasing with stored procedures
    8-5R
index time during modification
    10-20T
journal updates 10-33T
measurement 4-6T
multiple access arms per table 10-7T
network access 4-25T
nonsequential access 4-24T
of a LAN 12-5T
optimizing 4-3T to 4-40T
references to other books 4-3T
row access 4-22T
seek time 4-23T
sequential access 4-23T, 4-30T
sorting replaces nonsequential access
    4-36T
splitting tall tables 10-29T
splitting wide tables 10-28T
time costs of query 4-20T
use of derived data 10-30T
use of redundant data 10-31T
Performance analysis
    "80-20 rule" 4-7T
    measurement 4-6T
    methods 4-27T to 4-29T
    nonsequential access 4-36T to 4-39T
    optimizing techniques 4-4T
    setting up test environment 4-28T
    timing
        from 4GL program 4-6T
        from command script 4-6T
        from watch 4-6T
    using query plan 4-29T

verifying problem 4-4T
Permission
    with SYSTEM 8-71R
Phantom row 7-295R
PIPE keyword, in the OUTPUT
    statement 7-207R
Pipe symbol. *See* concatenation operator.
Plus (+) sign, arithmetic operator 7-356R
Populating tables 9-29T
PRECISION field
    description of 6-7R, 6-15R
    setting with SET DESCRIPTOR
        7-285R
    with GET DESCRIPTOR 7-162R
PREPARE statement
    description of 6-28T
    error return in SQLERRD 6-12T
    executing 7-135R
    increasing performance efficiency
        7-219R
    missing WHERE signalled 6-10T
    multiple SQL statements 6-29T
    multi-statement text 7-212R, 7-217R
    parameterizing a statement 7-213R
    parameterizing for SQL identifiers
        7-214R
    preparing GRANT 11-14T
    question (?) mark as placeholder
        7-208R
    releasing resources with FREE 7-157R
    restrictions with SELECT 7-211R
    statement identifier use 7-209R
    syntax 7-208R
    valid statement text 7-211R
    with concatenation operator 7-357R
Prepared statement
    describing returned values with
        DESCRIBE 7-119R
    executing 7-135R
    prepared object limit 7-209R
    valid statement text 7-211R
    See also PREPARE statement.
PREVIOUS keyword
    syntax in FETCH 7-144R
    use in FETCH 7-146R
Primary key constraint
    attribute depends on 8-22T
    composite 8-10T
    data type conversion 7-25R
    defining column as 7-79R

definition of 5-20T, 8-7T
dropping 7-29R
enforcing 7-71R
modifying a column with 7-24R
referencing 7-20R
requirements for 7-18R, 7-79R
restrictions with 8-7T
rules of use 7-28R, 7-80R

PRIMARY KEY keywords
in ALTER TABLE statement 7-18R
in CREATE TABLE 7-76R, 7-77R
use in ALTER TABLE 7-26R

Printing, specifying print program with
DBPRINT 4-12R

PRIOR keyword
syntax in FETCH 7-144R
use in FETCH 7-146R

Privilege
Alter 11-10T, 7-170R
and views 11-29T to 11-32T
automating grants of 11-13T
column-level 11-10T
Connect 11-6T, 7-167R
DBA 11-7T, 7-168R
default for stored procedures 8-15R
default for table using CREATE
TABLE 7-69R
Delete 11-9T, 11-29T, 7-170R
displaying 5-16T
displaying with the INFO statement
7-177R
encoded in system catalog 11-9T,
2-24R, 2-26R
granting 11-5T to 11-15T
Index 11-10T, 7-170R
Insert 11-9T, 11-29T, 7-170R
needed
to create a view 11-29T, 7-173R
to drop an index 7-128R
to modify data 5-15T, 7-170R
on a synonym 7-63R
on a view 11-29T, 7-92R
overview 1-8T
Resource 4-30T, 11-7T, 7-167R
Select 11-9T, 11-10T, 11-29T
Update 11-9T, 11-10T, 11-29T, 7-170R
when privileges conflict 7-166R
with DBA-privileged procedures
8-17R
with owner-privileged procedures
8-16R

See also Database-level privilege.
See also Table-level privilege.

PRIVILEGES FOR keywords, in INFO
statement 7-177R

PRIVILEGES keyword
syntax
in GRANT 7-169R
in REVOKE 7-237R
use
in GRANT 7-170R
in REVOKE 7-238R

Procedure name
conflict with function name 7-407R
naming conventions 7-407R

Procedure, stored. *See* Stored procedure.

Projection 2-7T

Promotable lock 7-20T, 7-23T

PUBLIC keyword
privilege granted to all users 11-6T
syntax
in GRANT 7-165R
in REVOKE 7-236R
use
in GRANT 7-168R
in REVOKE 7-237R

PUT statement
constant data with 7-11T
count of rows inserted 7-11T
sends returned data to buffer 7-10T
source of row values 7-222R
use in transactions 7-222R
with concatenation operator 7-357R
with FLUSH 7-222R

Q

Qualifier, field
for DATETIME 3-7R, 7-353R, 7-401R
for INTERVAL 3-12R, 7-398R, 7-404R

Query
improving performance of 4-27T to
4-39T
performance of 4-3T to 4-40T
piping results to another program
7-207R
sending results to an operating
system file 7-206R
sending results to another program
7-207R
stated in terms of data model 1-7T

S

SCALE field
  description of 6-7R, 6-15R
  setting with SET DESCRIPTOR
    7-285R
  with GET DESCRIPTOR 7-162R

Schema. *See* Data Model.

Screen dump, specifying file with
    DBSCREENOUT 4-14R

Scroll cursor
  active set 6-24T
  definition of 6-21T, 7-106R
  use of 7-106R
  with FETCH 7-146R
  with hold, in a transaction 7-296R

SCROLL keyword
  syntax in DECLARE 7-102R
  use in DECLARE 6-22T, 7-106R

Search condition. *See* Condition segment.

SECOND keyword
  syntax
    in DATETIME data type 7-353R
    in INTERVAL data type 7-398R
  use
    as DATETIME field qualifier 3-7R,
      7-401R
    as INTERVAL field qualifier
      3-12R, 7-404R

Security
  constraining inserted values 11-22T,
    11-27T
  database-level privileges 11-4T
  making database inaccessible 11-4T
  restricting access to columns 11-21T,
    11-22T
  restricting access to rows 11-21T,
    11-23T
  restricting access to view 11-29T
  table-level privileges 11-10T
  using host file system 11-4T
  using operating system facilities
    11-3T
  with stored procedures 11-3T

Seek time 4-23T

Select cursor
  closing 7-34R
  definition of 7-105R
  opening 7-198R
  reopening 7-199R
  use of 6-19T, 7-105R

SELECT keyword
  ambiguous use as procedure variable
    7-392R
  syntax
    in GRANT 7-169R
    in REVOKE 7-237R
  use
    in GRANT 7-170R
    in REVOKE 7-238R

Select list
  asterisk in 2-11T
  display label 2-41T
  expressions in 2-38T
  functions in 2-43T to 2-55T
  labels in 3-42T
  literal in 3-43T
  order of columns 2-12T
  selecting all columns 2-11T
  selecting specific columns 2-16T
  specifying a substring in 2-23T

Select privilege
  column level 11-10T
  definition of 11-9T, 7-170R
  with a view 11-29T

SELECT statement
  active set 2-24T
  aggregate functions in 2-44T, 7-375R
  alias names 2-67T
  as an argument to a stored procedure
    8-36R
  assigning values with 8-23R
  associating with cursor with
    DECLARE 7-105R
  asterisk in 2-11T
  column numbers 7-269R, 7-273R
  compound query 3-37T
  cursor for 6-18T, 6-19T
  date-oriented functions in 2-46T
  describing returned values with
    DESCRIBE 7-119R
  description of advanced 3-4T to 3-49T
  description of simple 2-3T to 2-70T
  display label 2-41T
  DISTINCT keyword 2-17T
  embedded 6-13T to 6-15T
  external tables 12-12T
  for joined tables 2-56T to 2-70T
  for single tables 2-10T to 2-55T
  FROM clause 7-258R
  functions 2-43T to 2-55T
  GROUP BY clause 7-268R

creating from an embedded language 8-7R
creating from DBACCESS 8-7R
cursors with 8-52R
DBA-privileged 8-15R
debugging 8-13R, 8-73R
definition of 8-5R
displaying contents 8-10R
displaying documentation 8-11R
documenting 8-8R
executing 8-11R
granting privileges on 7-171R
handling multiple rows 8-70R
header 8-27R, 8-40R
introduction to 8-5R
name confusion with SQL functions 8-22R
naming output file for TRACE statement 7-279R
owner-privileged 8-15R
receiving data from SELECT 7-253R
recursion 8-26R
returning values 8-27R
revoking privileges on 7-237R
security purposes 11-3T
simulating errors 8-67R
use 8-5R
variable 8-17R
Stored Procedure Language. *See* SPL.
stores5 database
call_type columns 1-7R
catalog table columns 1-6R
copying 11T, 14R
creating on INFORMIX-OnLine 11T, 14R
creating on INFORMIX-SE 11T, 15R
customer table columns 1-4R
cust_calls table columns 1-7R
data values 1-17R
description of 1-3R
items table columns 1-5R
manufact table columns 1-7R
map of 1-8R
orders table columns 1-5R
overview 10T, 13R
primary-foreign key relationships 1-10R to 1-17R
state table columns 1-8R
stock table columns 1-6R
structure of tables 1-3R

Structured Query Language. *See* SQL.
Subquery
beginning with ALL/ANY/SOME keywords 7-264R
beginning with EXISTS keyword 7-264R
beginning with IN keyword 7-263R
correlated 3-29T, 3-33T, 4-33T, 7-340R
definition of 7-260R
in Condition segment 7-340R
in DELETE statement 5-6T
in SELECT 3-29T to 3-37T
in UPDATE-SET 5-13T
in UPDATE-WHERE 5-12T
performance of 4-33T
restrictions with UNION operator 7-275R
restrictions with UPDATE 5-13T
with DISTINCT keyword 7-249R
See also Condition segment.
Subscripting 2-36T
Substring 2-23T
SUM function
as aggregate function 2-44T
syntax in expression 7-366R
use in expression 7-378R
Symbol table 10-27T
Synonym 12-12T to 12-14T
ANSI-compliant naming 7-63R
chaining 7-66R
chains of 12-14T
creating with CREATE SYNONYM 7-63R
difference from alias 7-63R
dropping 7-131R
Syntax diagram
conventions 5R
elements of 8R
System catalog
accessing 2-8R
database entries 7-42R
description of 2-3R
map of tables 2-27R
number of rows in a table 4-9T
privileges in 5-16T, 11-9T
querying 5-16T
sysblobs 2-10R
syschecks 2-10R
syscolauth 11-9T, 2-11R, 7-237R
syscoldepend 2-12R

VARCHAR data type
    considerations for UNLOAD
        statement 7-307R
    description of 9-16T, 3-17R
    effect on table size 10-14T
    requirements for LOAD statement
        7-191R
    syntax 7-351R
    used for performance 10-26T
    using as default value 7-16R, 7-74R
    with LENGTH function 2-53T
Variable
    default values in SPL 8-43R, 8-44R
    define in SPL 8-39R
    GLOBAL and LOCAL in SPL 8-18R
    global, in SPL 8-41R
    in SPL 8-17R
    local, in SPL 8-44R
    scope of SPL variable 8-40R
    unknown values in IF 8-56R
    with same name as a keyword 8-20R
View
    creating synonym for 7-63R
    deleting rows in 11-26T
    description of 11-21T
    dropped when basis is dropped
        11-24T
    dropping 7-134R
    effect of changing basis 11-24T
    effect on performance 4-29T
    inserting rows in 11-27T
    modifying 11-25T to 11-29T
    null inserted in unexposed columns
        11-27T
    privilege when accessing 11-29T
    privileges 11-29T to 11-32T, 7-92R
    privileges with GRANT 7-173R
    produces duplicate rows 11-23T
    restrictions on use 11-24T
    restrictions with UNION operator
        7-275R
    updating 7-312R
    updating duplicate rows 11-27T
    using CHECK OPTION 11-27T
    virtual column 11-26T, 7-93R
    with SELECT * notation 7-92R
View Name segment 7-420R
View, updatable 7-95R

W

WAIT keyword, in the SET LOCK
    MODE statement 7-297R
WARNING keyword, in the
    WHENEVER statement 7-323R,
    7-326R
Warnings
    with stored procedures 8-9R
WEEKDAY function
    as time function 2-46T, 2-50T
    syntax in expression 7-366R
WHENEVER statement, syntax and use
    7-323R
WHERE CURRENT OF keywords
    syntax
        in DELETE 7-116R
        in UPDATE 7-311R
    use
        in DELETE 7-7T
        in UPDATE 7-15T, 7-319R
WHERE keyword
    Boolean expression in 2-30T
    comparison condition 2-24T to 2-37T
    date-oriented functions in 2-50T
    enforcing data constraints 11-28T
    host variables in 6-13T
    in DELETE 5-4T to 5-6T
    joining tables 7-266R
    null data tests 2-30T
    prevents use of index 4-31T, 4-33T,
        4-34T
    range of values 2-27T
    relational operators 2-24T
    selecting rows 2-24T
    setting descriptions of items 7-281R
    subqueries in 3-29T
    syntax
        in DELETE 7-116R
        in SELECT 7-246R
        in UPDATE 7-311R
    testing a subscript 2-36T
    use
        in DELETE 7-117R
        in UPDATE 7-318R
        with a subquery 7-260R
        with ALL keyword 7-264R
        with ANY keyword 7-264R
        with BETWEEN keyword 7-261R
        with IN keyword 7-261R
        with IS keyword 7-262R