CORONEL / MORRIS / ROB

# DATABASE SYSTEMS

## Design, Implementation and Management

NINTH EDITION

7

SEVEN

# Introduction to Structured Query Language (SQL)

**In this chapter, you will learn:**

- The basic commands and functions of SQL
- How to use SQL for data administration (to create tables, indexes, and views)
- How to use SQL for data manipulation (to add, modify, delete, and retrieve data)
- How to use SQL to query a database for useful information

## Preview

In this chapter, you will learn the basics of Structured Query Language (SQL). SQL, pronounced S-Q-L by some and "sequel" by others, is composed of commands that enable users to create database and table structures, perform various types of data manipulation and data administration, and query the database to extract useful information. All relational DBMS software supports SQL, and many software vendors have developed extensions to the basic SQL command set.

Because SQL's vocabulary is simple, the language is relatively easy to learn. Its simplicity is enhanced by the fact that much of its work takes place behind the scenes. For example, a single command creates the complex table structures required to store and manipulate data successfully. Furthermore, SQL is a nonprocedural language; that is, the user specifies what must be done, but not how it is to be done. To issue SQL commands, end users and programmers do not need to know the physical data storage format or the complex activities that take place when a SQL command is executed.

Although quite useful and powerful, SQL is not meant to stand alone in the applications arena. Data entry with SQL is possible but awkward, as are data corrections and additions. SQL itself does not create menus, special report forms, overlays, pop-ups, or any of the other utilities and screen devices that end users usually expect. Instead, those features are available as vendor-supplied enhancements. SQL focuses on data definition (creating tables, indexes, and views) and data manipulation (adding, modifying, deleting, and retrieving data); we will cover these basic functions in this chapter. In spite of its limitations, SQL is a powerful tool for extracting information and managing data.

## 7.1 INTRODUCTION TO SQL

Ideally, a database language allows you to create database and table structures, to perform basic data management chores (add, delete, and modify), and to perform complex queries designed to transform the raw data into useful information. Moreover, a database language must perform such basic functions with minimal user effort, and its command structure and syntax must be easy to learn. Finally, it must be portable; that is, it must conform to some basic standard so that an individual does not have to relearn the basics when moving from one RDBMS to another. SQL meets those ideal database language requirements well.

SQL functions fit into two broad categories:

- It is a *data definition language (DDL)*: SQL includes commands to create database objects such as tables, indexes, and views, as well as commands to define access rights to those database objects. The data definition commands you will learn in this chapter are listed in Table 7.1.

**TABLE 7.1 SQL Data Definition Commands**

| COMMAND OR OPTION | DESCRIPTION |
|---|---|
| CREATE SCHEMA AUTHORIZATION | Creates a database schema |
| CREATE TABLE | Creates a new table in the user's database schema |
| NOT NULL | Ensures that a column will not have null values |
| UNIQUE | Ensures that a column will not have duplicate values |
| PRIMARY KEY | Defines a primary key for a table |
| FOREIGN KEY | Defines a foreign key for a table |
| DEFAULT | Defines a default value for a column (when no value is given) |
| CHECK | Validates data in an attribute |
| CREATE INDEX | Creates an index for a table |
| CREATE VIEW | Creates a dynamic subset of rows/columns from one or more tables |
| ALTER TABLE | Modifies a tables definition (adds, modifies, or deletes attributes or constraints) |
| CREATE TABLE AS | Creates a new table based on a query in the user's database schema |
| DROP TABLE | Permanently deletes a table (and its data) |
| DROP INDEX | Permanently deletes an index |
| DROP VIEW | Permanently deletes a view |

- It is a *data manipulation language (DML)*: SQL includes commands to insert, update, delete, and retrieve data within the database tables. The data manipulation commands you will learn in this chapter are listed in Table 7.2.

**TABLE 7.2 SQL Data Manipulation Commands**

| COMMAND OR OPTION | DESCRIPTION |
|---|---|
| INSERT | Inserts row(s) into a table |
| SELECT | Selects attributes from rows in one or more tables or views |
| WHERE | Restricts the selection of rows based on a conditional expression |
| GROUP BY | Groups the selected rows based on one or more attributes |
| HAVING | Restricts the selection of grouped rows based on a condition |
| ORDER BY | Orders the selected rows based on one or more attributes |
| UPDATE | Modifies an attribute's values in one or more table's rows |

**TABLE 7.2** SQL Data Manipulation Commands (continued)

| COMMAND OR OPTION | DESCRIPTION |
|---|---|
| DELETE | Deletes one or more rows from a table |
| COMMIT | Permanently saves data changes |
| ROLLBACK | Restores data to their original values |
| **COMPARISON OPERATORS** | |
| =, <, >, <=, >=, <> | Used in conditional expressions |
| **LOGICAL OPERATORS** | |
| AND/OR/NOT | Used in conditional expressions |
| **SPECIAL OPERATORS** | Used in conditional expressions |
| BETWEEN | Checks whether an attribute value is within a range |
| IS NULL | Checks whether an attribute value is null |
| LIKE | Checks whether an attribute value matches a given string pattern |
| IN | Checks whether an attribute value matches any value within a value list |
| EXISTS | Checks whether a subquery returns any rows |
| DISTINCT | Limits values to unique values |
| **AGGREGATE FUNCTIONS** | Used with SELECT to return mathematical summaries on columns |
| COUNT | Returns the number of rows with non-null values for a given column |
| MIN | Returns the minimum attribute value found in a given column |
| MAX | Returns the maximum attribute value found in a given column |
| SUM | Returns the sum of all values for a given column |
| AVG | Returns the average of all values for a given column |

You will be happy to know that SQL is relatively easy to learn. Its basic command set has a vocabulary of fewer than 100 words. Better yet, SQL is a nonprocedural language: you merely command *what* is to be done; you don't have to worry about *how* it is to be done. The American National Standards Institute (ANSI) prescribes a standard SQL—the current fully approved version is SQL-2003. The ANSI SQL standards are also accepted by the International Organization for Standardization (ISO), a consortium composed of national standards bodies of more than 150 countries. Although adherence to the ANSI/ISO SQL standard is usually required in commercial and government contract database specifications, many RDBMS vendors add their own special enhancements. Consequently, it is seldom possible to move a SQL-based application from one RDBMS to another without making some changes.

However, even though there are several different SQL "dialects," the differences among them are minor. Whether you use Oracle, Microsoft SQL Server, MySQL, IBM's DB2, Microsoft Access, or any other well-established RDBMS, a software manual should be sufficient to get you up to speed if you know the material presented in this chapter.

At the heart of SQL is the query. In Chapter 1, Database Systems, you learned that a query is a spur-of-the-moment question. Actually, in the SQL environment, the word *query* covers both questions and actions. Most SQL queries are used to answer questions such as these: "What products currently held in inventory are priced over $100, and what is the quantity on hand for each of those products?" "How many employees have been hired since January 1, 2008 by each of the company's departments?" However, many SQL queries are used to perform actions such as adding or deleting table rows or changing attribute values within tables. Still other SQL queries create new tables or indexes. In short, for a DBMS, a query is simply a SQL statement that must be executed. But before you can use SQL to query a database, you must define the database environment for SQL with its data definition commands.

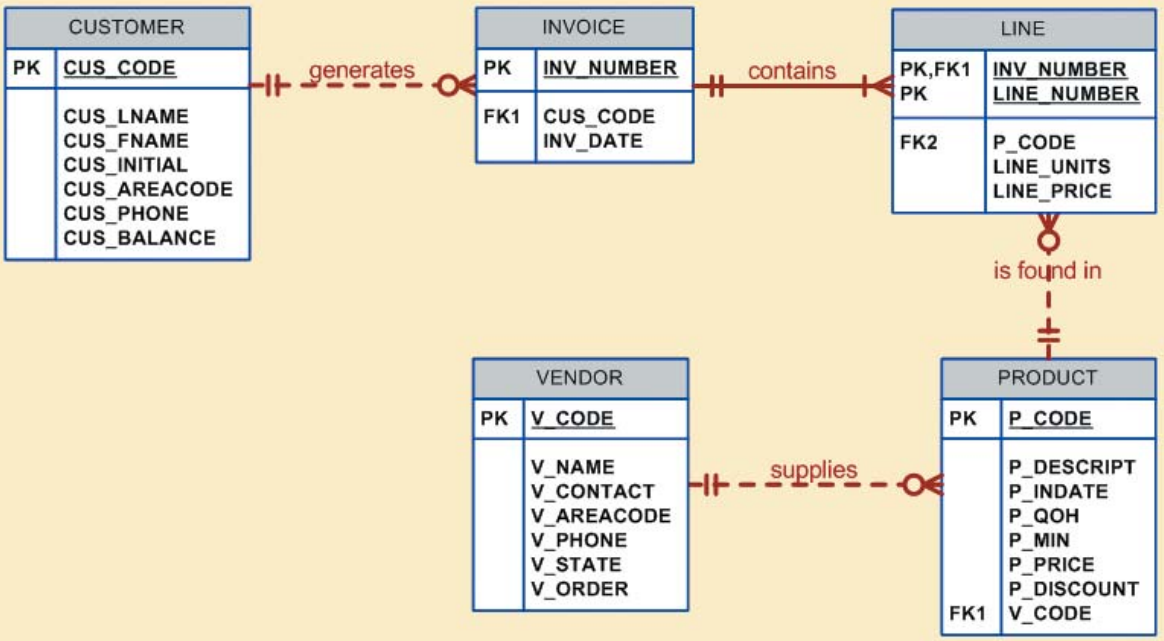
## 7.2 DATA DEFINITION COMMANDS

Before examining the SQL syntax for creating and defining tables and other elements, let's first examine the simple database model and the database tables that will form the basis for the many SQL examples you'll explore in this chapter.

### 7.2.1 The Database Model

A simple database composed of the following tables is used to illustrate the SQL commands in this chapter: CUSTOMER, INVOICE, LINE, PRODUCT, and VENDOR. This database model is shown in Figure 7.1.

**FIGURE 7.1 The database model**




The database model in Figure 7.1 reflects the following business rules:

- A customer may generate many invoices. Each invoice is generated by one customer.
- An invoice contains one or more invoice lines. Each invoice line is associated with one invoice.
- Each invoice line references one product. A product may be found in many invoice lines. (You can sell more than one hammer to more than one customer.)
- A vendor *may* supply many products. Some vendors do not (yet?) supply products. (For example, a vendor list may include *potential* vendors.)
- If a product is vendor-supplied, that product is supplied by only a single vendor.
- Some products are not supplied by a vendor. (For example, some products may be produced in-house or bought on the open market.)

As you can see in Figure 7.1, the database model contains many tables. However, to illustrate the initial set of data definition commands, the focus of attention will be the PRODUCT and VENDOR tables. You will have the opportunity to use the remaining tables later in this chapter and in the problem section.

## ONLINE CONTENT

The database model in Figure 7.1 is implemented in the Microsoft Access **Ch07_SaleCo** database located in the Premium Website for this book. (This database contains a few additional tables that are not reflected in Figure 7.1. These tables are used for discussion purposes only.) If you use MS Access, you can use the database supplied online. However, it is strongly suggested that you create your own database structures so you can practice the SQL commands illustrated in this chapter.

SQL script files for creating the tables and loading the data in Oracle and MS SQL Server are also located in the Premium Website. How you connect to your database depends on how the software was installed on your computer. Follow the instructions provided by your instructor or school.

So that you have a point of reference for understanding the effect of the SQL queries, the contents of the PRODUCT and VENDOR tables are listed in Figure 7.2.

**FIGURE 7.2** **The VENDOR and PRODUCT tables**

**Table name: VENDOR** **Database name: Ch07_SaleCo**

| V_CODE | V_NAME | V_CONTACT | V_AREACODE | V_PHONE | V_STATE | V_ORDER |
|---|---|---|---|---|---|---|
| 21225 | Bryson, Inc. | Smithson | 615 | 223-3234 | TN | Y |
| 21226 | SuperLoo, Inc. | Flushing | 904 | 215-8995 | FL | N |
| 21231 | D&E Supply | Singh | 615 | 228-3245 | TN | Y |
| 21344 | Gomez Bros. | Ortega | 615 | 889-2546 | KY | N |
| 22567 | Dome Supply | Smith | 901 | 678-1419 | GA | N |
| 23119 | Randsets Ltd. | Anderson | 901 | 678-3998 | GA | Y |
| 24004 | Brackman Bros. | Browning | 615 | 228-1410 | TN | N |
| 24288 | ORDVA, Inc. | Hakford | 615 | 898-1234 | TN | Y |
| 25443 | B&K, Inc. | Smith | 904 | 227-0093 | FL | N |
| 25501 | Damal Supplies | Smythe | 615 | 890-3529 | TN | N |
| 25595 | Rubicon Systems | Orton | 904 | 456-0092 | FL | Y |

**Table name: PRODUCT**

| P_CODE | P_DESCRIPT | P_INDATE | P_QOH | P_MIN | P_PRICE | P_DISCOUNT | V_CODE |
|---|---|---|---|---|---|---|---|
| 11QER/31 | Power painter, 15 psi., 3-nozzle | 03-Nov-09 | 8 | 5 | 109.99 | 0.00 | 25595 |
| 13-Q2/P2 | 7.25-in. pwr. saw blade | 13-Dec-09 | 32 | 15 | 14.99 | 0.05 | 21344 |
| 14-Q1/L3 | 9.00-in. pwr. saw blade | 13-Nov-09 | 18 | 12 | 17.49 | 0.00 | 21344 |
| 1546-QQ2 | Hrd. cloth, 1/4-in., 2x50 | 15-Jan-10 | 15 | 8 | 39.95 | 0.00 | 23119 |
| 1558-QW1 | Hrd. cloth, 1/2-in., 3x50 | 15-Jan-10 | 23 | 5 | 43.99 | 0.00 | 23119 |
| 2232/QTY | B&D jigsaw, 12-in. blade | 30-Dec-09 | 8 | 5 | 109.92 | 0.05 | 24288 |
| 2232/QWE | B&D jigsaw, 8-in. blade | 24-Dec-09 | 6 | 5 | 99.87 | 0.05 | 24288 |
| 2238/QPD | B&D cordless drill, 1/2-in. | 20-Jan-10 | 12 | 5 | 38.95 | 0.05 | 25595 |
| 23109-HB | Claw hammer | 20-Jan-10 | 23 | 10 | 9.95 | 0.10 | 21225 |
| 23114-AA | Sledge hammer, 12 lb. | 02-Jan-10 | 8 | 5 | 14.40 | 0.05 | |
| 54778-2T | Rat-tail file, 1/8-in. fine | 15-Dec-09 | 43 | 20 | 4.99 | 0.00 | 21344 |
| 89-WRE-Q | Hicut chain saw, 16 in. | 07-Feb-10 | 11 | 5 | 256.99 | 0.05 | 24288 |
| PVC23DRT | PVC pipe, 3.5-in., 8-ft | 20-Feb-10 | 188 | 75 | 5.87 | 0.00 | |
| SM-18277 | 1.25-in. metal screw, 25 | 01-Mar-10 | 172 | 75 | 6.99 | 0.00 | 21225 |
| SW-23116 | 2.5-in. wd. screw, 50 | 24-Feb-10 | 237 | 100 | 8.45 | 0.00 | 21231 |
| WR3/TT3 | Steel matting, 4'x8'x1/6", .5" mesh | 17-Jan-10 | 18 | 5 | 119.95 | 0.10 | 25595 |

Note the following about these tables. (The features correspond to the business rules reflected in the ERD shown in Figure 7.1.)

- The VENDOR table contains vendors who are not referenced in the PRODUCT table. Database designers note that possibility by saying that PRODUCT is *optional* to VENDOR; a vendor may exist without a reference to a product. You examined such optional relationships in detail in Chapter 4, Entity Relationship (ER) Modeling.

- Existing V_CODE values in the PRODUCT table must (and do) have a match in the VENDOR table to ensure referential integrity.
- A few products are supplied factory-direct, a few are made in-house, and a few may have been bought in a warehouse sale. In other words, a product is not necessarily supplied by a vendor. Therefore, VENDOR is optional to PRODUCT.

A few of the conditions just described were made for the sake of illustrating specific SQL features. For example, null V_CODE values were used in the PRODUCT table to illustrate (later) how you can track such nulls using SQL.

### 7.2.2 Creating the Database

Before you can use a new RDBMS, you must complete two tasks: first, create the database structure, and second, create the tables that will hold the end-user data. To complete the first task, the RDBMS creates the physical files that will hold the database. When you create a new database, the RDBMS automatically creates the data dictionary tables in which to store the metadata and creates a default database administrator. Creating the physical files that will hold the database means interacting with the operating system and the file systems supported by the operating system. Therefore, creating the database structure is the one feature that tends to differ substantially from one RDBMS to another. The good news is that it is relatively easy to create a database structure, regardless of which RDBMS you use.

If you use Microsoft Access, creating the database is simple: start Access, select *File* → *New* → *Blank Database*, specify the folder in which you want to store the database, and then name the database. However, if you work in a database environment typically used by larger organizations, you will probably use an enterprise RDBMS such as Oracle, SQL Server, MySQL, or DB2. Given their security requirements and greater complexity, those database products require a more elaborate database creation process. (See Appendix N, Creating a New Database using Oracle 11g, for an illustration of specific instructions to create a database structure in Oracle.)

You will be relieved to discover that, *with the exception of the database creation process*, most RDBMS vendors use SQL that deviates little from the ANSI standard SQL. For example, most RDBMSs require that each SQL command ends with a semicolon. However, some SQL implementations do not use a semicolon. Important syntax differences among implementations will be highlighted in the Note boxes.

If you are using an enterprise RDBMS, before you can start creating tables you must be authenticated by the RDBMS. **Authentication** is the process through which the DBMS verifies that only registered users may access the database. To be authenticated, you must log on to the RDBMS using a user ID and a password created by the database administrator. In an enterprise RDBMS, every user ID is associated with a database schema.

### 7.2.3 The Database Schema

In the SQL environment, a **schema** is a group of database objects—such as tables and indexes—that are related to each other. Usually, the schema belongs to a single user or application. A single database can hold multiple schemas belonging to different users or applications. Think of a schema as a logical grouping of database objects, such as tables, indexes, and views. Schemas are useful in that they group tables by owner (or function) and enforce a first level of security by allowing each user to see only the tables that belong to that user.

ANSI SQL standards define a command to create a database schema:

CREATE SCHEMA AUTHORIZATION {creator};

Therefore, if the creator is JONES, use the command:

CREATE SCHEMA AUTHORIZATION JONES;

Most enterprise RDBMSs support that command. However, the command is seldom used directly—that is, from the command line. (When a user is created, the DBMS automatically assigns a schema to that user.) When the DBMS *is* used, the CREATE SCHEMA AUTHORIZATION command must be issued by the user who owns the schema. That is, if you log on as JONES, you can only use CREATE SCHEMA AUTHORIZATION JONES.

For most RDBMSs, the CREATE SCHEMA AUTHORIZATION is optional. That is why this chapter focuses on the ANSI SQL commands required to create and manipulate tables.

### 7.2.4 Data Types

In the data dictionary in Table 7.3, note particularly the data types selected. Keep in mind that data-type selection is usually dictated by the nature of the data and by the intended use. For example:

- P_PRICE clearly requires some kind of numeric data type; defining it as a character field is not acceptable.
- Just as clearly, a vendor name is an obvious candidate for a character data type. For example, VARCHAR2(35) fits well because vendor names are "variable-length" character strings, and in this case, such strings may be up to 35 characters long.
- At first glance, it might seem logical to select a numeric data type for V_AREACODE because it contains only digits. However, adding and subtracting area codes does not yield meaningful results. Therefore, selecting a character data type is more appropriate. This is true for many common attributes found in business data models. For example, even though zip codes contain all digits, they must be defined as character data because some zip codes begin with the digit zero (0), and a numeric data type would cause the leading zero to be dropped.
- U.S. state abbreviations are always two characters, so CHAR(2) is a logical choice.
- Selecting P_INDATE to be a (Julian) DATE field rather than a character field is desirable because the Julian dates allow you to make simple date comparisons and to perform date arithmetic. For instance, if you have used DATE fields, you can determine how many days there are between them.

If you use DATE fields, you can also determine what the date will be in say, 60 days from a given P_INDATE by using P_INDATE + 60. Or you can use the RDBMS's system date—SYSDATE in Oracle, GETDATE() in MS SQL Server, and Date() in Access—to determine the answer to questions such as, "What will be the date 60 days from today?" For example, you might use SYSDATE + 60 (in Oracle), GETDATE() + 60 (in MS SQL Server), or Date() + 60 (in Access).

Date arithmetic capability is particularly useful in billing. Perhaps you want your system to start charging interest on a customer balance 60 days after the invoice is generated. Such simple date arithmetic would be impossible if you used a character data type.

Data-type selection sometimes requires professional judgment. For example, you must make a decision about the V_CODE's data type as follows:

- If you want the computer to generate new vendor codes by adding 1 to the largest recorded vendor code, you must classify V_CODE as a numeric attribute. (You cannot perform mathematical procedures on character data.) The designation INTEGER will ensure that only the counting numbers (integers) can be used. Most SQL implementations also permit the use of SMALLINT for integer values up to six digits.
- If you do not want to perform mathematical procedures based on V_CODE, you should classify it as a character attribute, even though it is composed entirely of numbers. Character data are "quicker" to process in queries. Therefore, when there is no need to perform mathematical procedures on the attribute, store it as a character attribute.

The first option is used to demonstrate the SQL procedures in this chapter.

**TABLE 7.3** Data Dictionary for the CH07_SALECO Database

| TABLE NAME | ATTRIBUTE NAME | CONTENTS | TYPE | FORMAT | RANGE* | REQUIRED | PK OR FK | FK REFERENCED TABLE |
|---|---|---|---|---|---|---|---|---|
| PRODUCT | P_CODE | Product code | CHAR(10) | XXXXXXXXX | NA | Y | PK | |
| | P_DESCRIPT | Product description | VARCHAR(35) | Xxxxxxxxxxx | NA | Y | | |
| | P_INDATE | Stocking date | DATE | DD-MON-YYYY | NA | Y | | |
| | P_QOH | Units available | SMALLINT | #### | 0-9999 | Y | | |
| | P_MIN | Minimum units | SMALLINT | #### | 0-9999 | Y | | |
| | P_PRICE | Product price | NUMBER(8,2) | ####.## | 0.00-9999.00 | Y | | |
| | P_DISCOUNT | Discount rate | NUMBER(5,2) | 0.## | 0.00-0.20 | Y | | |
| | V_CODE | Vendor code | INTEGER | ### | 100-999 | | FK | VENDOR |
| | | | | | | | | |
| VENDOR | V_CODE | Vendor code | INTEGER | ##### | 1000-9999 | Y | PK | |
| | V_NAME | Vendor name | CHAR(35) | Xxxxxxxxxxxxx | NA | Y | | |
| | V_CONTACT | Contact person | CHAR(25) | Xxxxxxxxxxxxx | NA | Y | | |
| | V_AREACODE | Area code | CHAR(3) | 999 | NA | Y | | |
| | V_PHONE | Phone number | CHAR(8) | 999-9999 | NA | Y | | |
| | V_STATE | State | CHAR(2) | XX | NA | Y | | |
| | V_ORDER | Previous order | CHAR(1) | X | Y or N | Y | | |

FK = Foreign key
PK = Primary key
CHAR = Fixed character length data, 1 to 255 characters
VARCHAR = Variable character length data, 1 to 2,000 characters. VARCHAR is automatically converted to VARCHAR2 in Oracle.
NUMBER = Numeric data. NUMBER(9,2) is used to specify numbers with two decimal places and up to nine digits long, including the decimal places. Some RDBMSs permit the use of a MONEY or a CURRENCY data type.
INT = Integer values only
SMALLINT = Small integer values only
DATE formats vary. Commonly accepted formats are: 'DD-MON-YYYY', 'DD-MON-YY', 'MM/DD/YYYY', and 'MM/DD/YY'
* Not all the ranges shown here will be illustrated in this chapter. However, you can use these constraints to practice writing your own constraints.

When you define the attribute's data type, you must pay close attention to the expected use of the attributes for sorting and data-retrieval purposes. For example, in a real estate application, an attribute that represents the numbers of bathrooms in a home (H_BATH_NUM) could be assigned the CHAR(3) data type because it is highly unlikely the application will do any addition, multiplication, or division with the number of bathrooms. Based on the CHAR(3) data-type definition, valid H_BATH_NUM values would be '2','1','2.5','10'. However, this data-type decision creates potential problems. For example, if an application sorts the homes by number of bathrooms, a query would "see" the value '10' as less than '2', which is clearly incorrect. So you must give some thought to the expected use of the data in order to properly define the attribute data type.

The data dictionary in Table 7.3 contains only a few of the data types supported by SQL. For teaching purposes, the selection of data types is limited to ensure that almost any RDBMS can be used to implement the examples. If your RDBMS is fully compliant with ANSI SQL, it will support many more data types than the ones shown in Table 7.4. And many RDBMSs support data types beyond the ones specified in ANSI SQL.

**TABLE 7.4 Some Common SQL Data Types**

| DATA TYPE | FORMAT | COMMENTS |
|---|---|---|
| **Numeric** | NUMBER(L,D) | The declaration NUMBER(7,2) indicates numbers that will be stored with two decimal places and may be up to seven digits long, including the sign and the decimal place. Examples: 12.32, -134.99. |
| | INTEGER | May be abbreviated as INT. Integers are (whole) counting numbers, so they cannot be used if you want to store numbers that require decimal places. |
| | SMALLINT | Like INTEGER but limited to integer values up to six digits. If your integer values are relatively small, use SMALLINT instead of INT. |
| | DECIMAL(L,D) | Like the NUMBER specification, but the storage length is a *minimum specification. That is, greater lengths are acceptable, but smaller ones are not. DECIMAL(9,2), DECIMAL(9), and DECIMAL are all acceptable.* |
| **Character** | CHAR(L) | Fixed-length character data for up to 255 characters. If you store strings that are not as long as the CHAR parameter value, the remaining spaces are left unused. Therefore, if you specify CHAR(25), strings such as Smith and Katzenjammer are each stored as 25 characters. However, a U.S. area code is always three digits long, so CHAR(3) would be appropriate if you wanted to store such codes. |
| | VARCHAR(L) or VARCHAR2(L) | Variable-length character data. The designation VARCHAR2(25) will let you store characters up to 25 characters long. However, VARCHAR will not leave unused spaces. Oracle automatically converts VARCHAR to VARCHAR2. |
| **Date** | DATE | Stores dates in the Julian date format. |

In addition to the data types shown in Table 7.4, SQL supports several other data types, including TIME, TIMESTAMP, REAL, DOUBLE, FLOAT, and intervals such as INTERVAL DAY TO HOUR. Many RDBMSs have also expanded the list to include other types of data, such as LOGICAL, CURRENCY, AutoNumber (Access), and sequence (Oracle). However, because this chapter is designed to introduce the SQL basics, the discussion is limited to the data types summarized in Table 7.4.

### 7.2.5 Creating Table Structures

Now you are ready to implement the PRODUCT and VENDOR table structures with the help of SQL, using the **CREATE TABLE** syntax shown next.

```
CREATE TABLE tablename (
        column1          data type       [constraint] [,
        column2          data type       [constraint] ] [,
        PRIMARY KEY      (column1        [, column2]) ] [,
        FOREIGN KEY      (column1        [, column2]) REFERENCES tablename] [,
        CONSTRAINT       constraint ] );
```

**Online Content**

All the SQL commands you will see in this chapter are located in script files in the Premium Website for this book. You can copy and paste the SQL commands into your SQL program. Script files are provided for Oracle and SQL Server users.

To make the SQL code more readable, most SQL programmers use one line per column (attribute) definition. In addition, spaces are used to line up the attribute characteristics and constraints. Finally, both table and attribute names are fully capitalized. Those conventions are used in the following examples that create VENDOR and PRODUCT tables and throughout the book.

**Note**

SQL SYNTAX

Syntax notation for SQL commands used in this book:

| | |
|---|---|
| CAPITALS | Required SQL command keywords |
| *italics* | An end-user-provided parameter (generally required) |
| {a \| b \| ..} | A mandatory parameter; use one option from the list separated by \| |
| [.....] | An optional parameter—anything inside square brackets is optional |
| *Tablename* | The name of a table |
| *Column* | The name of an attribute in a table |
| *data type* | A valid data-type definition |
| *constraint* | A valid constraint definition |
| *condition* | A valid conditional expression (evaluates to true or false) |
| *columnlist* | One or more column names or expressions separated by commas |
| *tablelist* | One or more table names separated by commas |
| *conditionlist* | One or more conditional expressions separated by logical operators |
| *expression* | A simple value (such as 76 or Married) or a formula (such as P_PRICE − 10) |

```
CREATE TABLE VENDOR (
V_CODE          INTEGER         NOT NULL   UNIQUE,
V_NAME          VARCHAR(35)     NOT NULL,
V_CONTACT       VARCHAR(15)     NOT NULL,
V_AREACODE      CHAR(3)         NOT NULL,
V_PHONE         CHAR(8)         NOT NULL,
V_STATE         CHAR(2)         NOT NULL,
V_ORDER         CHAR(1)         NOT NULL,
PRIMARY KEY (V_CODE));
```

**NOTE**

- Because the PRODUCT table contains a foreign key that references the VENDOR table, create the VENDOR table first. (In fact, the M side of a relationship always references the 1 side. Therefore, in a 1:M relationship, you must *always* create the table for the 1 side first.)
- If your RDBMS does not support the VARCHAR2 and FCHAR format, use CHAR.
- Oracle accepts the VARCHAR data type and automatically converts it to VARCHAR2.
- If your RDBMS does not support SINT or SMALLINT, use INTEGER or INT. If INTEGER is not supported, use NUMBER.
- If you use Access, you can use the NUMBER data type, but you cannot use the number delimiters at the SQL level. For example, using NUMBER(8,2) to indicate numbers with up to eight characters and two decimal places is fine in Oracle, but you cannot use it in Access—you must use NUMBER without the delimiters.
- If your RDBMS does not support primary and foreign key designations or the UNIQUE specification, delete them from the SQL code shown here.
- If you use the PRIMARY KEY designation in Oracle, you do not need the NOT NULL and UNIQUE specifications.
- The ON UPDATE CASCADE clause is part of the ANSI standard, but it may not be supported by your RDBMS. In that case, delete the ON UPDATE CASCADE clause.

```
CREATE TABLE PRODUCT (
P_CODE          VARCHAR(10)     NOT NULL   UNIQUE,
P_DESCRIPT      VARCHAR(35)     NOT NULL,
P_INDATE        DATE            NOT NULL,
P_QOH           SMALLINT        NOT NULL,
P_MIN           SMALLINT        NOT NULL,
P_PRICE         NUMBER(8,2)     NOT NULL,
P_DISCOUNT      NUMBER(5,2)     NOT NULL,
V_CODE          INTEGER,
PRIMARY KEY (P_CODE),
FOREIGN KEY (V_CODE) REFERENCES VENDOR ON UPDATE CASCADE);
```

As you examine the preceding SQL table-creating command sequences, note the following features:

- The NOT NULL specifications for the attributes ensure that a data entry will be made. When it is crucial to have the data available, the NOT NULL specification will not allow the end user to leave the attribute empty (with no data entry at all). Because this specification is made at the table level and stored in the data dictionary, application programs can use this information to create the data dictionary validation automatically.
- The UNIQUE specification creates a unique index in the respective attribute. Use it to avoid having duplicated values in a column.

- The primary key attributes contain both a NOT NULL and a UNIQUE specification. Those specifications enforce the entity integrity requirements. If the NOT NULL and UNIQUE specifications are not supported, use PRIMARY KEY without the specifications. (For example, if you designate the PK in MS Access, the NOT NULL and UNIQUE specifications are automatically assumed and are not spelled out.)
- The entire table definition is enclosed in parentheses. A comma is used to separate each table element (attributes, primary key, and foreign key) definition.

**NOTE**

If you are working with a composite primary key, all of the primary key's attributes are contained within the parentheses and are separated with commas. For example, the LINE table in Figure 7.1 has a primary key that consists of the two attributes INV_NUMBER and LINE_NUMBER. Therefore, you would define the primary key by typing:

```
PRIMARY KEY (INV_NUMBER, LINE_NUMBER),
```

*The order of the primary key components is important* because the indexing starts with the first-mentioned attribute, then proceeds with the next attribute, and so on. In this example, the line numbers would be ordered within each of the invoice numbers:

| INV_NUMBER | LINE_NUMBER |
|---|---|
| 1001 | 1 |
| 1001 | 2 |
| 1002 | 1 |
| 1003 | 1 |
| 1003 | 2 |

- The ON UPDATE CASCADE specification ensures that if you make a change in any VENDOR's V_CODE, that change is automatically applied to all foreign key references throughout the system (cascade) to ensure that referential integrity is maintained. (Although the ON UPDATE CASCADE clause is part of the ANSI standard, some RDBMSs, such as Oracle, do not support ON UPDATE CASCADE. If your RDBMS does not support the clause, delete it from the code shown here.)
- An RDBMS will automatically enforce referential integrity for foreign keys. That is, you cannot have an invalid entry in the foreign key column; at the same time, you cannot delete a vendor row as long as a product row references that vendor.
- The command sequence ends with a semicolon. (Remember, your RDBMS may require that you omit the semicolon.)

**NOTE**

NOTE ABOUT COLUMN NAMES

Do *not* use mathematical symbols such as +, −, and / in your column names; instead, use an underscore to separate words, if necessary. For example, PER-NUM might generate an error message, but PER_NUM is acceptable. Also, do *not* use reserved words. **Reserved words** are words used by SQL to perform specific functions. For example, in some RDBMSs, the column name INITIAL will generate the message invalid column name.

**NOTE**

NOTE TO ORACLE USERS

When you press the Enter key after typing each line, a line number is automatically generated as long as you do not type a semicolon before pressing the Enter key. For example, Oracle's execution of the CREATE TABLE command will look like this:

```
CREATE TABLE PRODUCT (
  2     P_CODE          VARCHAR2(10)
  3     CONSTRAINT      PRODUCT_P_CODE_PK PRIMARY KEY,
  4     P_DESCRIPT      VARCHAR2(35)      NOT NULL,
  5     P_INDATE        DATE              NOT NULL,
  6     P_QOH           NUMBER            NOT NULL,
  7     P_MIN           NUMBER            NOT NULL,
  8     P_PRICE         NUMBER(8,2)       NOT NULL,
  9     P_DISCOUNT      NUMBER(5,2)       NOT NULL,
 10     V_CODE           NUMBER,
 11     CONSTRAINT      PRODUCT_V_CODE_FK
 12     FOREIGN KEY     V_CODE REFERENCES VENDOR)
 13     ;
```

In the preceding SQL command sequence, note the following:

- The attribute definition for P_CODE starts in line 2 and ends with a comma at the end of line 3.
- The CONSTRAINT clause (line 3) allows you to define and name a constraint in Oracle. You can name the constraint to meet your own naming conventions. In this case, the constraint was named PRODUCT_P_CODE_PK.
- Examples of constraints are NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, and CHECK. For additional details about constraints, see below.
- To define a PRIMARY KEY constraint, you could also use the following syntax: P_CODE VARCHAR2(10) PRIMARY KEY,.
- In this case, Oracle would automatically name the constraint.
- Lines 11 and 12 define a FOREIGN KEY constraint name PRODUCT_V_CODE_FK for the attribute V_CODE. The CONSTRAINT clause is generally used at the end of the CREATE TABLE command sequence.
- *If you do not name the constraints yourself, Oracle will automatically assign a name. Unfortunately, the Oracle-assigned name makes sense only to Oracle, so you will have a difficult time deciphering it later. You should assign a name that makes sense to human beings!*

### 7.2.6 SQL Constraints

In Chapter 3, The Relational Database Model, you learned that adherence to rules on entity integrity and referential integrity is crucial in a relational database environment. Fortunately, most SQL implementations support both integrity rules. Entity integrity is enforced automatically when the primary key is specified in the CREATE TABLE command sequence. For example, you can create the VENDOR table structure and set the stage for the enforcement of entity integrity rules by using:

```
PRIMARY KEY (V_CODE)
```

In the PRODUCT table's CREATE TABLE sequence, note that referential integrity has been enforced by specifying in the PRODUCT table:

```
FOREIGN KEY (V_CODE) REFERENCES VENDOR ON UPDATE CASCADE
```

That foreign key constraint definition ensures that:

- You cannot delete a vendor from the VENDOR table if at least one product row references that vendor. This is the default behavior for the treatment of foreign keys.
- On the other hand, if a change is made in an existing VENDOR table's V_CODE, that change must be reflected automatically in any PRODUCT table V_CODE reference (ON UPDATE CASCADE). That restriction makes it impossible for a V_CODE value to exist in the PRODUCT table pointing to a nonexistent VENDOR table V_CODE value. In other words, the ON UPDATE CASCADE specification ensures the preservation of referential integrity. (Oracle does not support ON UPDATE CASCADE.)

In general, ANSI SQL permits the use of ON DELETE and ON UPDATE clauses to cover CASCADE, SET NULL, or SET DEFAULT.

**ONLINE CONTENT**

For a more detailed discussion of the options for the ON DELETE and ON UPDATE clauses, see **Appendix D, Converting an ER Model into a Database Structure,** Section D.2, General Rules Governing Relationships Among Tables. Appendix D is in the Premium Website.

**NOTE**

NOTE ABOUT REFERENTIAL CONSTRAINT ACTIONS

The support for the referential constraints actions varies from product to product. For example:

- MS Access, SQL Server, and Oracle support ON DELETE CASCADE.
- MS Access and SQL Server support ON UPDATE CASCADE.
- Oracle does not support ON UPDATE CASCADE.
- Oracle supports SET NULL.
- MS Access and SQL Server do not support SET NULL.
- Refer to your product manuals for additional information on referential constraints.

While MS Access does not support ON DELETE CASCADE or ON UPDATE CASCADE at the SQL command-line level, it does support them through the relationship window interface. In fact, whenever you try to establish a relationship between two tables in Access, the relationship window interface will automatically pop up.

Besides the PRIMARY KEY and FOREIGN KEY constraints, the ANSI SQL standard also defines the following constraints:

- The NOT NULL constraint ensures that a column does not accept nulls.
- The UNIQUE constraint ensures that all values in a column are unique.
- The DEFAULT constraint assigns a value to an attribute when a new row is added to a table. The end user may, of course, enter a value other than the default value.
- The CHECK constraint is used to validate data when an attribute value is entered. The CHECK constraint does precisely what its name suggests: it checks to see that a specified condition exists. Examples of such constraints include the following:
  - *The minimum order value must be at least 10.*
  - *The date must be after April 15, 2010.*

If the CHECK constraint is met for the specified attribute (that is, the condition is true), the data are accepted for that attribute. If the condition is found to be false, an error message is generated and the data are not accepted.

Note that the CREATE TABLE command lets you define constraints in two different places:

- When you create the column definition (known as a *column constraint*).
- When you use the CONSTRAINT keyword (known as a *table constraint*).

A column constraint applies to just one column; a table constraint may apply to many columns. Those constraints are supported at varying levels of compliance by enterprise RDBMSs.

In this chapter, Oracle is used to illustrate SQL constraints. For example, note that the following SQL command sequence uses the DEFAULT and CHECK constraints to define the table named CUSTOMER.

```
CREATE TABLE CUSTOMER (
CUS_CODE            NUMBER          PRIMARY KEY,
CUS_LNAME           VARCHAR(15)     NOT NULL,
CUS_FNAME           VARCHAR(15)     NOT NULL,
CUS_INITIAL         CHAR(1),
CUS_AREACODE        CHAR(3)         DEFAULT '615'       NOT NULL
                                    CHECK(CUS_AREACODE IN ('615','713','931')),
CUS_PHONE           CHAR(8)         NOT NULL,
CUS_BALANCE         NUMBER(9,2)     DEFAULT 0.00,
CONSTRAINT CUS_UI1 UNIQUE (CUS_LNAME, CUS_FNAME));
```

In this case, the CUS_AREACODE attribute is assigned a default value of '615'. Therefore, if a new CUSTOMER table row is added and the end user makes no entry for the area code, the '615' value will be recorded. Also note that the CHECK condition restricts the values for the customer's area code to 615, 713, and 931; any other values will be rejected.

It is important to note that the DEFAULT value applies only when new rows are added to a table and then only when no value is entered for the customer's area code. (The default value is not used when the table is modified.) In contrast, the CHECK condition is validated whether a customer row is added *or modified*. However, while the CHECK condition may include any valid expression, it applies only to the attributes in the table being checked. If you want to check for conditions that include attributes in other tables, you must use triggers. (See Chapter 8, Advanced SQL.) Finally, the last line of the CREATE TABLE command sequence creates a unique index constraint (named CUS_UI1) on the customer's last name and first name. The index will prevent the entry of two customers with the same last name and first name. (This index merely illustrates the process. Clearly, it should be possible to have more than one person named John Smith in the CUSTOMER table.)

**NOTE**

NOTE TO MS ACCESS USERS

MS Access does not accept the DEFAULT or CHECK constraints. However, MS Access will accept the CONSTRAINT CUS_UI1 UNIQUE (CUS_LNAME, CUS_FNAME) line and create the unique index.

In the following SQL command to create the INVOICE table, the DEFAULT constraint assigns a default date to a new invoice, and the CHECK constraint validates that the invoice date is greater than January 1, 2010.

```
CREATE TABLE INVOICE (
INV_NUMBER          NUMBER          PRIMARY KEY,
CUS_CODE            NUMBER          NOT NULL REFERENCES CUSTOMER(CUS_CODE),
INV_DATE            DATE            DEFAULT SYSDATE NOT NULL,
CONSTRAINT INV_CK1 CHECK (INV_DATE > TO_DATE('01-JAN-2010','DD-MON-YYYY')));
```

In this case, notice the following:

- The CUS_CODE attribute definition contains REFERENCES CUSTOMER (CUS_CODE) to indicate that the CUS_CODE is a foreign key. This is another way to define a foreign key.
- The DEFAULT constraint uses the SYSDATE special function. This function always returns today's date.
- The invoice date (INV_DATE) attribute is automatically given today's date (returned by SYSDATE) when a new row is added and no value is given for the attribute.
- A CHECK constraint is used to validate that the invoice date is greater than 'January 1, 2010'. When comparing a date to a manually entered date in a CHECK clause, Oracle requires the use of the TO_DATE function. The TO_DATE function takes two parameters: the literal date and the date format used.

The final SQL command sequence creates the LINE table. The LINE table has a composite primary key (INV_NUMBER, LINE_NUMBER) and uses a UNIQUE constraint in INV_NUMBER and P_CODE to ensure that the same product is not ordered twice in the same invoice.

```
CREATE TABLE LINE (
INV_NUMBER          NUMBER          NOT NULL,
LINE_NUMBER         NUMBER(2,0)     NOT NULL,
P_CODE              VARCHAR(10)     NOT NULL,
LINE_UNITS          NUMBER(9,2)     DEFAULT 0.00        NOT NULL,
LINE_PRICE          NUMBER(9,2)     DEFAULT 0.00        NOT NULL,
PRIMARY KEY (INV_NUMBER, LINE_NUMBER),
FOREIGN KEY (INV_NUMBER) REFERENCES INVOICE ON DELETE CASCADE,
FOREIGN KEY (P_CODE) REFERENCES PRODUCT(P_CODE),
CONSTRAINT LINE_UI1 UNIQUE(INV_NUMBER, P_CODE));
```

In the creation of the LINE table, note that a UNIQUE constraint is added to prevent the duplication of an invoice line. A UNIQUE constraint is enforced through the creation of a unique index. Also note that the ON DELETE CASCADE foreign key action enforces referential integrity. The use of ON DELETE CASCADE is recommended for weak entities to ensure that the deletion of a row in the strong entity automatically triggers the deletion of the corresponding rows in the dependent weak entity. In that case, the deletion of an INVOICE row will automatically delete all of the LINE rows related to the invoice. In the following section, you will learn more about indexes and how to use SQL commands to create them.

### 7.2.7 SQL Indexes

You learned in Chapter 3 that indexes can be used to improve the efficiency of searches and to avoid duplicate column values. In the previous section, you saw how to declare unique indexes on selected attributes when the table is created. In fact, when you declare a primary key, the DBMS automatically creates a unique index. Even with this feature, you often need additional indexes. The ability to create indexes quickly and efficiently is important. Using the **CREATE INDEX** command, SQL indexes can be created on the basis of any selected attribute. The syntax is:

CREATE [UNIQUE] INDEX *indexname* ON *tablename*(*column1* [, *column2*])

For example, based on the attribute P_INDATE stored in the PRODUCT table, the following command creates an index named P_INDATEX:

```
CREATE INDEX P_INDATEX ON PRODUCT(P_INDATE);
```

SQL does not let you write over an existing index without warning you first, thus preserving the index structure within the data dictionary. Using the UNIQUE index qualifier, you can even create an index that prevents you from using a value that has been used before. Such a feature is especially useful when the index attribute is a candidate key whose values must not be duplicated:

## 7.3 DATA MANIPULATION COMMANDS

In this section, you will learn how to use the basic SQL data manipulation commands INSERT, SELECT, COMMIT, UPDATE, ROLLBACK, and DELETE.

### 7.3.1 Adding Table Rows

SQL requires the use of the **INSERT** command to enter data into a table. The INSERT command's basic syntax looks like this:

INSERT INTO *tablename* VALUES (*value1, value2, ... , valuen*)

Because the PRODUCT table uses its V_CODE to reference the VENDOR table's V_CODE, an integrity violation will occur if those VENDOR table V_CODE values don't yet exist. Therefore, you need to enter the VENDOR rows before the PRODUCT rows. Given the VENDOR table structure defined earlier and the sample VENDOR data shown in Figure 7.2, you would enter the first two data rows as follows:

```
INSERT INTO VENDOR
      VALUES (21225,'Bryson, Inc.','Smithson','615','223-3234','TN','Y');
INSERT INTO VENDOR
      VALUES (21226,'Superloo, Inc.','Flushing','904','215-8995','FL','N');
```

and so on, until all of the VENDOR table records have been entered.

(To see the contents of the VENDOR table, use the SELECT * FROM VENDOR; command.)

The PRODUCT table rows would be entered in the same fashion, using the PRODUCT data shown in Figure 7.2. For example, the first two data rows would be entered as follows, pressing the Enter key at the end of each line:

```
INSERT INTO PRODUCT
      VALUES ('11QER/31','Power painter, 15 psi., 3-nozzle','03-Nov-09',8,5,109.99,0.00,25595);
INSERT INTO PRODUCT
      VALUES ('13-Q2/P2','7.25-in. pwr. saw blade','13-Dec-09',32,15,14.99, 0.05, 21344);
```

(To see the contents of the PRODUCT table, use the SELECT * FROM PRODUCT; command.)

**NOTE**

Date entry is a function of the date format expected by the DBMS. For example, March 25, 2010 might be shown as 25-Mar-2010 in Access and Oracle, or it might be displayed in other presentation formats in another RDBMS. MS Access requires the use of # delimiters when performing any computations or comparisons based on date attributes, as in P_INDATE >= #25-Mar-10#.

In the preceding data entry lines, observe that:

- The row contents are entered between parentheses. Note that the first character after VALUES is a parenthesis and that the last character in the command sequence is also a parenthesis.
- Character (string) and date values must be entered between apostrophes (').
- Numerical entries are *not* enclosed in apostrophes.
- Attribute entries are separated by commas.
- A value is required for each column in the table.

This version of the INSERT commands adds one table row at a time.

## Inserting Rows with Null Attributes

Thus far, you have entered rows in which all of the attribute values are specified. But what do you do if a product does not have a vendor or if you don't yet know the vendor code? In those cases, you would want to leave the vendor code null. To enter a null, use the following syntax:

```
INSERT INTO PRODUCT
      VALUES ('BRT-345','Titanium drill bit','18-Oct-09', 75, 10, 4.50, 0.06, NULL);
```

Incidentally, note that the NULL entry is accepted only because the V_CODE attribute is optional—the NOT NULL declaration was not used in the CREATE TABLE statement for this attribute.

## Inserting Rows with Optional Attributes

There might be occasions when more than one attribute is optional. Rather than declaring each attribute as NULL in the INSERT command, you can indicate just the attributes that have required values. You do that by listing the attribute names inside parentheses after the table name. For the purpose of this example, assume that the only required attributes for the PRODUCT table are P_CODE and P_DESCRIPT:

```
INSERT INTO PRODUCT(P_CODE, P_DESCRIPT) VALUES ('BRT-345','Titanium drill bit');
```

### 7.3.2 Saving Table Changes

Any changes made to the table contents are not saved on disk until you close the database, close the program you are using, or use the **COMMIT** command. If the database is open and a power outage or some other interruption occurs before you issue the COMMIT command, your changes will be lost and only the original table contents will be retained. The syntax for the COMMIT command is:

```
COMMIT [WORK]
```

The COMMIT command permanently saves *all* changes—such as rows added, attributes modified, and rows deleted—made to any table in the database. Therefore, if you intend to make your changes to the PRODUCT table permanent, it is a good idea to save those changes by using:

```
COMMIT;
```

**NOTE**

NOTE TO MS ACCESS USERS
MS Access doesn't support the COMMIT command because it automatically saves changes after the execution of each SQL command.

However, the COMMIT command's purpose is not just to save changes. In fact, the ultimate purpose of the COMMIT and ROLLBACK commands (see Section 7.3.5) is to ensure database update integrity in transaction management. (You will see how such issues are addressed in Chapter 10, Transaction Management and Concurrency Control.)

### 7.3.3 Listing Table Rows

The **SELECT** command is used to list the contents of a table. The syntax of the SELECT command is as follows:

SELECT *columnlist* FROM *tablename*

The *columnlist* represents one or more attributes, separated by commas. You could use the * (asterisk) as a wildcard character to list all attributes. A **wildcard character** is a symbol that can be used as a general substitute for other characters or commands. For example, to list all attributes and all rows of the PRODUCT table, use:

SELECT * FROM PRODUCT;

Figure 7.3 shows the output generated by that command. (Figure 7.3 shows all of the rows in the PRODUCT table that serve as the basis for subsequent discussions. If you entered only the PRODUCT table's first two records, as shown in the preceding section, the output of the preceding SELECT command would show only the rows you entered. Don't worry about the difference between your SELECT output and the output shown in Figure 7.3. When you complete the work in this section, you will have created and populated your VENDOR and PRODUCT tables with the correct rows for use in future sections.)

**FIGURE 7.3** **The contents of the PRODUCT table**

| P_CODE | P_DESCRIPT | P_INDATE | P_QOH | P_MIN | P_PRICE | P_DISCOUNT | V_CODE |
|---|---|---|---|---|---|---|---|
| 11QER/31 | Power painter, 15 psi., 3-nozzle | 03-Nov-09 | 8 | 5 | 109.99 | 0.00 | 25595 |
| 13-Q2/P2 | 7.25-in. pwr. saw blade | 13-Dec-09 | 32 | 15 | 14.99 | 0.05 | 21344 |
| 14-Q1/L3 | 9.00-in. pwr. saw blade | 13-Nov-09 | 18 | 12 | 17.49 | 0.00 | 21344 |
| 1546-QQ2 | Hrd. cloth, 1/4-in., 2x50 | 15-Jan-10 | 15 | 8 | 39.95 | 0.00 | 23119 |
| 1558-QW1 | Hrd. cloth, 1/2-in., 3x50 | 15-Jan-10 | 23 | 5 | 43.99 | 0.00 | 23119 |
| 2232/QTY | B&D jigsaw, 12-in. blade | 30-Dec-09 | 8 | 5 | 109.92 | 0.05 | 24288 |
| 2232/QWE | B&D jigsaw, 8-in. blade | 24-Dec-09 | 6 | 5 | 99.87 | 0.05 | 24288 |
| 2238/QPD | B&D cordless drill, 1/2-in. | 20-Jan-10 | 12 | 5 | 38.95 | 0.05 | 25595 |
| 23109-HB | Claw hammer | 20-Jan-10 | 23 | 10 | 9.95 | 0.10 | 21225 |
| 23114-AA | Sledge hammer, 12 lb. | 02-Jan-10 | 8 | 5 | 14.40 | 0.05 | |
| 54778-2T | Rat-tail file, 1/8-in. fine | 15-Dec-09 | 43 | 20 | 4.99 | 0.00 | 21344 |
| 89-WRE-Q | Hicut chain saw, 16 in. | 07-Feb-10 | 11 | 5 | 256.99 | 0.05 | 24288 |
| PVC23DRT | PVC pipe, 3.5-in., 8-ft | 20-Feb-10 | 188 | 75 | 5.87 | 0.00 | |
| SM-18277 | 1.25-in. metal screw, 25 | 01-Mar-10 | 172 | 75 | 6.99 | 0.00 | 21225 |
| SW-23116 | 2.5-in. wd. screw, 50 | 24-Feb-10 | 237 | 100 | 8.45 | 0.00 | 21231 |
| WR3/TT3 | Steel matting, 4'x8'x1/6", .5" mesh | 17-Jan-10 | 18 | 5 | 119.95 | 0.10 | 25595 |

**NOTE**

Your listing may not be in the order shown in Figure 7.3. The listings shown in the figure are the result of system-controlled primary-key-based index operations. You will learn later how to control the output so that it conforms to the order you have specified.

**NOTE**

NOTE TO ORACLE USERS

Some SQL implementations (such as Oracle's) cut the attribute labels to fit the width of the column. However, Oracle lets you set the width of the display column to show the complete attribute name. You can also change the display format, regardless of how the data are stored in the table. For example, if you want to display dollar symbols and commas in the P_PRICE output, you can declare:

COLUMN P_PRICE FORMAT $99,999.99

to change the output 12347.67 to $12,347.67.

In the same manner, to display only the first 12 characters of the P_DESCRIPT attribute, use:

COLUMN P_DESCRIPT FORMAT A12 TRUNCATE

Although SQL commands can be grouped together on a single line, complex command sequences are best shown on separate lines, with space between the SQL command and the command's components. Using that formatting convention makes it much easier to see the components of the SQL statements, making it easy to trace the SQL logic, and if necessary, to make corrections. The number of spaces used in the indention is up to you. For example, note the following format for a more complex statement:

```
SELECT    P_CODE, P_DESCRIPT, P_INDATE, P_QOH, P_MIN, P_PRICE, P_DISCOUNT, V_CODE
FROM      PRODUCT;
```

When you run a SELECT command on a table, the RDBMS returns a set of one or more rows that have the same characteristics as a relational table. In addition, the SELECT command lists all rows from the table you specified in the FROM clause. This is a very important characteristic of SQL commands. By default, most SQL data manipulation commands operate over an entire table (or relation). That is why SQL commands are said to be *set-oriented* commands. A SQL set-oriented command works over a set of rows. The set may include one or more columns and zero or more rows from one or more tables.

### 7.3.4 Updating Table Rows

Use the **UPDATE** command to modify data in a table. The syntax for this command is:

```
UPDATE    tablename
SET       columnname = expression [, columnname = expression]
[WHERE    conditionlist ];
```

For example, if you want to change P_INDATE from December 13, 2009, to January 18, 2010, in the second row of the PRODUCT table (see Figure 7.3), use the primary key (13-Q2/P2) to locate the correct (second) row. Therefore, type:

```
UPDATE    PRODUCT
SET       P_INDATE = '18-JAN-2010'
WHERE     P_CODE = '13-Q2/P2';
```

If more than one attribute is to be updated in the row, separate the corrections with commas:

```
UPDATE    PRODUCT
SET       P_INDATE = '18-JAN-2010', P_PRICE = 17.99, P_MIN = 10
WHERE     P_CODE = '13-Q2/P2';
```

What would have happened if the previous UPDATE command had not included the WHERE condition? The P_INDATE, P_PRICE, and P_MIN values would have been changed in *all* rows of the PRODUCT table. Remember, the UPDATE command is a set-oriented operator. Therefore, if you don't specify a WHERE condition, the UPDATE command will apply the changes to *all* rows in the specified table.

Confirm the correction(s) by using this SELECT command to check the PRODUCT table's listing:

```
SELECT * FROM PRODUCT;
```

### 7.3.5 Restoring Table Contents

If you have not yet used the COMMIT command to store the changes permanently in the database, you can restore the database to its previous condition with the **ROLLBACK** command. ROLLBACK undoes any changes since the last COMMIT command and brings the data back to the values that existed before the changes were made. To restore the data to their "prechange" condition, type:

```
ROLLBACK;
```

and then press the Enter key. Use the SELECT statement again to see that the ROLLBACK did, in fact, restore the data to their original values.

COMMIT and ROLLBACK work only with data manipulation commands that are used to add, modify, or delete table rows. For example, assume that you perform these actions:

1. CREATE a table called SALES.
2. INSERT 10 rows in the SALES table.
3. UPDATE two rows in the SALES table.
4. Execute the ROLLBACK command.

Will the SALES table be removed by the ROLLBACK command? No, the ROLLBACK command will undo *only* the results of the INSERT and UPDATE commands. All data definition commands (CREATE TABLE) are automatically committed to the data dictionary and cannot be rolled back. The COMMIT and ROLLBACK commands are examined in greater detail in Chapter 10.

**NOTE**

NOTE TO MS ACCESS USERS
MS Access does not support the ROLLBACK command.

Some RDBMSs, such as Oracle, automatically COMMIT data changes when issuing data definition commands. For example, if you had used the CREATE INDEX command after updating the two rows in the previous example, all previous changes would have been committed automatically; doing a ROLLBACK afterward wouldn't have undone anything. *Check your RDBMS manual to understand these subtle differences.*

### 7.3.6 DELETING TABLE ROWS

It is easy to delete a table row using the **DELETE** statement; the syntax is:

```
DELETE FROM     tablename
[WHERE          conditionlist ];
```

For example, if you want to delete from the PRODUCT table the product that you added earlier whose code (P_CODE) is 'BRT-345', use:

```
DELETE FROM     PRODUCT
WHERE           P_CODE = 'BRT-345';
```

In that example, the primary key value lets SQL find the exact record to be deleted. However, deletions are not limited to a primary key match; any attribute may be used. For example, in your PRODUCT table, you will see that there are several products for which the P_MIN attribute is equal to 5. Use the following command to delete all rows from the PRODUCT table for which the P_MIN is equal to 5:

```
DELETE FROM     PRODUCT
WHERE           P_MIN = 5;
```

Check the PRODUCT table's contents again to verify that all products with P_MIN equal to 5 have been deleted.

Finally, remember that DELETE is a set-oriented command. And keep in mind that the WHERE condition is optional. Therefore, if you do not specify a WHERE condition, *all* rows from the specified table will be deleted!

### 7.3.7 Inserting Table Rows with a Select Subquery

You learned in Section 7.3.1 how to use the INSERT statement to add rows to a table. In that section, you added rows one at a time. In this section, you will learn how to add multiple rows to a table, using another table as the source of the data. The syntax for the INSERT statement is:

INSERT INTO *tablename* SELECT *columnlist* FROM *tablename*;

In that case, the INSERT statement uses a SELECT subquery. A **subquery**, also known as a **nested query** or an **inner query**, is a query that is embedded (or nested) inside another query. The inner query is always executed first by the RDBMS. Given the previous SQL statement, the INSERT portion represents the outer query, and the SELECT portion represents the subquery. You can nest queries (place queries inside queries) many levels deep; in every case, the output of the inner query is used as the input for the outer (higher-level) query. In Chapter 8 you will learn more about the various types of subqueries.

The values returned by the SELECT subquery should match the attributes and data types of the table in the INSERT statement. If the table into which you are inserting rows has one date attribute, one number attribute, and one character attribute, the SELECT subquery should return one or more rows in which the first column has date values, the second column has number values, and the third column has character values.

**Online Content**

Before you execute the commands in the following sections, you **MUST** do the following:

- If you are using Oracle, run the **sqlintrodbinit.sql** script file in the Premium Website to create all tables and load the data in the database.
- If you are using Access, copy the original **Ch07_SaleCo.mbd** file from the Premium Website.

## 7.4 SELECT QUERIES

In this section, you will learn how to fine-tune the SELECT command by adding restrictions to the search criteria. SELECT, coupled with appropriate search conditions, is an incredibly powerful tool that enables you to transform data into information. For example, in the following sections, you will learn how to create queries that can be used to answer questions such as these: "What products were supplied by a particular vendor?" "Which products are priced below $10?" "How many products supplied by a given vendor were sold between January 5, 2010 and March 20, 2010?"

### 7.4.1 Selecting Rows with Conditional Restrictions

You can select partial table contents by placing restrictions on the rows to be included in the output. This is done by using the WHERE clause to add conditional restrictions to the SELECT statement. The following syntax enables you to specify which rows to select:

SELECT *columnlist*
FROM *tablelist*
[WHERE *conditionlist* ];

The SELECT statement retrieves all rows that match the specified condition(s)—also known as the *conditional criteria*—you specified in the WHERE clause. The *conditionlist* in the WHERE clause of the SELECT statement is represented by one or more conditional expressions, separated by logical operators. The WHERE clause is optional.

If no rows match the specified criteria in the WHERE clause, you see a blank screen or a message that tells you that no rows were retrieved. For example, the query:

```
SELECT    P_DESCRIPT, P_INDATE, P_PRICE, V_CODE
FROM      PRODUCT
WHERE     V_CODE = 21344;
```

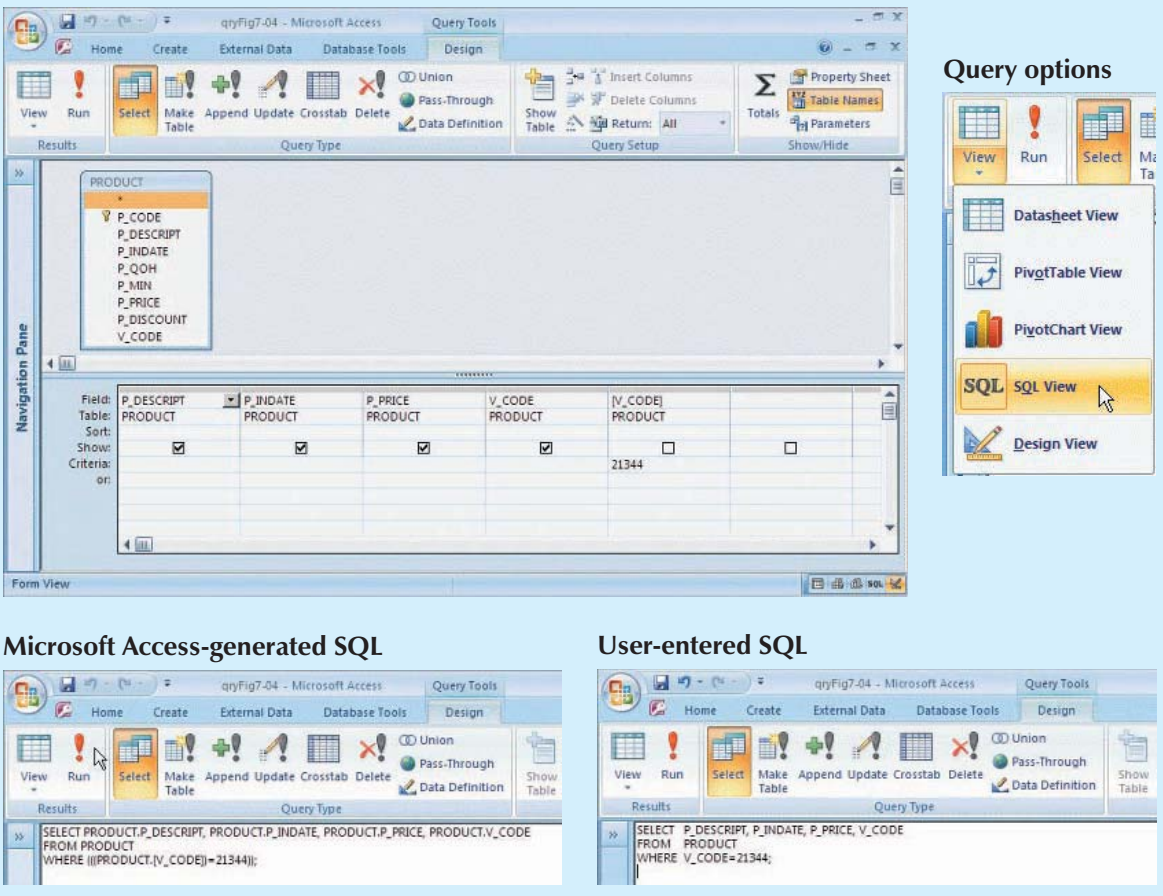
returns the description, date, and price of products with a vendor code of 21344, as shown in Figure 7.4.

**FIGURE 7.4 Selected PRODUCT table attributes for VENDOR code 21344**

| P_DESCRIPT | P_INDATE | P_PRICE | V_CODE |
|---|---|---|---|
| 7.25-in. pwr. saw blade | 13-Dec-09 | 14.99 | 21344 |
| 9.00-in. pwr. saw blade | 13-Nov-09 | 17.49 | 21344 |
| Rat-tail file, 1/8-in. fine | 15-Dec-09 | 4.99 | 21344 |

MS Access users can use the Access QBE (query by example) query generator. Although the Access QBE generates its own "native" version of SQL, you can also elect to type standard SQL in the Access SQL window, as shown at the bottom of Figure 7.5. Figure 7.5 shows the Access QBE screen, the SQL window's QBE-generated SQL, and the listing of the modified SQL.

**FIGURE 7.5 The Microsoft Access QBE and its SQL**




Numerous conditional restrictions can be placed on the selected table contents. For example, the comparison operators shown in Table 7.6 can be used to restrict output.

**NOTE**

NOTE TO MS ACCESS USERS

The MS Access QBE interface automatically designates the data source by using the table name as a prefix. You will discover later that the table name prefix is used to avoid ambiguity when the same column name appears in multiple tables. For example, both the VENDOR and the PRODUCT tables contain the V_CODE attribute. Therefore, if both tables are used (as they would be in a join), the source of the V_CODE attribute must be specified.

**TABLE 7.6 Comparison Operators**

| SYMBOL | MEANING |
|---|---|
| = | Equal to |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| <> or != | Not equal to |

The following example uses the "not equal to" operator:

```
SELECT    P_DESCRIPT, P_INDATE, P_PRICE, V_CODE
FROM      PRODUCT
WHERE     V_CODE <> 21344;
```

The output, shown in Figure 7.6, lists all of the rows for which the vendor code is *not* 21344.

Note that, in Figure 7.6, rows with nulls in the V_CODE column (see Figure 7.3) are not included in the SELECT command's output.

**FIGURE 7.6 Selected PRODUCT table attributes for VENDOR codes other than 21344**

| P_DESCRIPT | P_INDATE | P_PRICE | V_CODE |
|---|---|---|---|
| Power painter, 15 psi., 3-nozzle | 03-Nov-09 | 109.99 | 25595 |
| Hrd. cloth, 1/4-in., 2x50 | 15-Jan-10 | 39.95 | 23119 |
| Hrd. cloth, 1/2-in., 3x50 | 15-Jan-10 | 43.99 | 23119 |
| B&D jigsaw, 12-in. blade | 30-Dec-09 | 109.92 | 24288 |
| B&D jigsaw, 8-in. blade | 24-Dec-09 | 99.87 | 24288 |
| B&D cordless drill, 1/2-in. | 20-Jan-10 | 38.95 | 25595 |
| Claw hammer | 20-Jan-10 | 9.95 | 21225 |
| Hicut chain saw, 16 in. | 07-Feb-10 | 256.99 | 24288 |
| 1.25-in. metal screw, 25 | 01-Mar-10 | 6.99 | 21225 |
| 2.5-in. wd. screw, 50 | 24-Feb-10 | 8.45 | 21231 |
| Steel matting, 4'x8'x1/6", .5" mesh | 17-Jan-10 | 119.95 | 25595 |

**FIGURE 7.7 Selected PRODUCT table attributes with a P_PRICE restriction**

| P_DESCRIPT | P_QOH | P_MIN | P_PRICE |
|---|---|---|---|
| Claw hammer | 23 | 10 | 9.95 |
| Rat-tail file, 1/8-in. fine | 43 | 20 | 4.99 |
| PVC pipe, 3.5-in., 8-ft | 188 | 75 | 5.87 |
| 1.25-in. metal screw, 25 | 172 | 75 | 6.99 |
| 2.5-in. wd. screw, 50 | 237 | 100 | 8.45 |

The command sequence:

```
SELECT    P_DESCRIPT, P_QOH, P_MIN, P_PRICE
FROM      PRODUCT
WHERE     P_PRICE <= 10;
```

yields the output shown in Figure 7.7.

## Using Comparison Operators on Character Attributes

Because computers identify all characters by their (numeric) American Standard Code for Information Interchange (ASCII) codes, comparison operators may even be used to place restrictions on character-based attributes. Therefore, the command:

```
SELECT    P_CODE, P_DESCRIPT, P_QOH, P_MIN,
          P_PRICE
FROM      PRODUCT
WHERE     P_CODE < '1558-QW1';
```

would be correct and would yield a list of all rows in which the P_CODE is alphabetically less than 1558-QW1. (Because the

ASCII code value for the letter *B* is greater than the value of the letter *A*, it follows that *A* is less than *B*.) Therefore, the output will be generated as shown in Figure 7.8.

**FIGURE 7.8 Selected PRODUCT table attributes: the ASCII code effect**

| P_CODE | P_DESCRIPT | P_QOH | P_MIN | P_PRICE |
|---|---|---|---|---|
| 11QER/31 | Power painter, 15 psi., 3-nozzle | 8 | 5 | 109.99 |
| 13-Q2/P2 | 7.25-in. pwr. saw blade | 32 | 15 | 14.99 |
| 14-Q1/L3 | 9.00-in. pwr. saw blade | 18 | 12 | 17.49 |
| 1546-QQ2 | Hrd. cloth, 1/4-in., 2x50 | 15 | 8 | 39.95 |

String (character) comparisons are made from left to right. This left-to-right comparison is especially useful when attributes such as names are to be compared. For example, the string "Ardmore" would be judged *greater than* the string "Aarenson" but *less than* the string "Brown"; such results may be used to generate alphabetical listings like those found in a phone directory. If the characters 0–9 are stored as strings, the same left-to-right string comparisons can lead to apparent anomalies. For example, the ASCII code for the character "5" is, as expected, *greater than* the ASCII code for the character "4." Yet the same "5" will also be judged *greater than* the string "44" because the *first* character in the string "44" is less than the string "5." For that reason, you may get some unexpected results from comparisons when dates or other numbers are stored in character format. This also applies to date comparisons. For example, the left-to-right ASCII character comparison would force the conclusion that the date "01/01/2010" occurred *before* "12/31/2009." Because the leftmost character "0" in "01/01/2010" is *less than* the leftmost character "1" in "12/31/2009," "01/01/2010" is *less than* "12/31/2009." Naturally, if date strings are stored in a yyyy/mm/dd format, the comparisons will yield appropriate results, but this is a nonstandard date presentation. That's why all current RDBMSs support "date" data types; you should use them. In addition, using "date" data types gives you the benefit of date arithmetic.

## Using Comparison Operators on Dates

Date procedures are often more software-specific than other SQL procedures. For example, the query to list all of the rows in which the inventory stock dates occur on or after January 20, 2010 will look like this:

```
SELECT    P_DESCRIPT, P_QOH, P_MIN, P_PRICE, P_INDATE
FROM      PRODUCT
WHERE     P_INDATE >= '20-Jan-2010';
```

(Remember that MS Access users must use the # delimiters for dates. For example, you would use #20-Jan-10# in the above WHERE clause.) The date-restricted output is shown in Figure 7.9.

**FIGURE 7.9 Selected PRODUCT table attributes: date restriction**

| P_DESCRIPT | P_QOH | P_MIN | P_PRICE | P_INDATE |
|---|---|---|---|---|
| B&D cordless drill, 1/2-in. | 12 | 5 | 38.95 | 20-Jan-10 |
| Claw hammer | 23 | 10 | 9.95 | 20-Jan-10 |
| Hicut chain saw, 16 in. | 11 | 5 | 256.99 | 07-Feb-10 |
| PVC pipe, 3.5-in., 8-ft | 188 | 75 | 5.87 | 20-Feb-10 |
| 1.25-in. metal screw, 25 | 172 | 75 | 6.99 | 01-Mar-10 |
| 2.5-in. wd. screw, 50 | 237 | 100 | 8.45 | 24-Feb-10 |

## Using Computed Columns and Column Aliases

Suppose that you want to determine the total value of each of the products currently held in inventory. Logically, that determination requires the multiplication of each product's quantity on hand by its current price. You can accomplish this task with the following command:

```
SELECT    P_DESCRIPT, P_QOH, P_PRICE, P_QOH *
          P_PRICE
FROM      PRODUCT;
```

**FIGURE 7.10** SELECT statement with a computed column

| P_DESCRIPT | P_QOH | P_PRICE | Expr1 |
|---|---|---|---|
| Power painter, 15 psi., 3-nozzle | 8 | 109.99 | 879.92 |
| 7.25-in. pwr. saw blade | 32 | 14.99 | 479.68 |
| 9.00-in. pwr. saw blade | 18 | 17.49 | 314.82 |
| Hrd. cloth, 1/4-in., 2x50 | 15 | 39.95 | 599.25 |
| Hrd. cloth, 1/2-in., 3x50 | 23 | 43.99 | 1011.77 |
| B&D jigsaw, 12-in. blade | 8 | 109.92 | 879.36 |
| B&D jigsaw, 8-in. blade | 6 | 99.87 | 599.22 |
| B&D cordless drill, 1/2-in. | 12 | 38.95 | 467.40 |
| Claw hammer | 23 | 9.95 | 228.85 |
| Sledge hammer, 12 lb. | 8 | 14.40 | 115.20 |
| Rat-tail file, 1/8-in. fine | 43 | 4.99 | 214.57 |
| Hicut chain saw, 16 in. | 11 | 256.99 | 2826.89 |
| PVC pipe, 3.5-in., 8-ft | 188 | 5.87 | 1103.56 |
| 1.25-in. metal screw, 25 | 172 | 6.99 | 1202.28 |
| 2.5-in. wd. screw, 50 | 237 | 8.45 | 2002.65 |
| Steel matting, 4'x8'x1/6", .5" mesh | 18 | 119.95 | 2159.10 |

Entering that SQL command in Access generates the output shown in Figure 7.10.

SQL accepts any valid expressions (or formulas) in the computed columns. Such formulas can contain any valid mathematical operators and functions that are applied to attributes in any of the tables specified in the FROM clause of the SELECT statement. Note also that Access automatically adds an Expr label to all computed columns. (The first computed column would be labeled Expr1; the second, Expr2; and so on.) Oracle uses the actual formula text as the label for the computed column.

To make the output more readable, the SQL standard permits the use of aliases for any column in a SELECT statement. An **alias** is an alternative name given to a column or table in any SQL statement.

For example, you can rewrite the previous SQL statement as:

```
SELECT    P_DESCRIPT, P_QOH, P_PRICE, P_QOH * P_PRICE AS TOTVALUE
FROM      PRODUCT;
```

The output of that command is shown in Figure 7.11.

**FIGURE 7.11** SELECT statement with a computed column and an alias

| P_DESCRIPT | P_QOH | P_PRICE | TOTVALUE |
|---|---|---|---|
| Power painter, 15 psi., 3-nozzle | 8 | 109.99 | 879.92 |
| 7.25-in. pwr. saw blade | 32 | 14.99 | 479.68 |
| 9.00-in. pwr. saw blade | 18 | 17.49 | 314.82 |
| Hrd. cloth, 1/4-in., 2x50 | 15 | 39.95 | 599.25 |
| Hrd. cloth, 1/2-in., 3x50 | 23 | 43.99 | 1011.77 |
| B&D jigsaw, 12-in. blade | 8 | 109.92 | 879.36 |
| B&D jigsaw, 8-in. blade | 6 | 99.87 | 599.22 |
| B&D cordless drill, 1/2-in. | 12 | 38.95 | 467.40 |
| Claw hammer | 23 | 9.95 | 228.85 |
| Sledge hammer, 12 lb. | 8 | 14.40 | 115.20 |
| Rat-tail file, 1/8-in. fine | 43 | 4.99 | 214.57 |
| Hicut chain saw, 16 in. | 11 | 256.99 | 2826.89 |
| PVC pipe, 3.5-in., 8-ft | 188 | 5.87 | 1103.56 |
| 1.25-in. metal screw, 25 | 172 | 6.99 | 1202.28 |
| 2.5-in. wd. screw, 50 | 237 | 8.45 | 2002.65 |
| Steel matting, 4'x8'x1/6", .5" mesh | 18 | 119.95 | 2159.10 |

You could also use a computed column, an alias, and date arithmetic in a single query. For example, assume that you want to get a list of out-of-warranty products that have been stored more than 90 days. In that case, the P_INDATE is at least 90 days less than the current (system) date. The MS Access version of this query is:

```
SELECT    P_CODE, P_INDATE, DATE() - 90 AS
          CUTDATE
FROM      PRODUCT
WHERE     P_INDATE <= DATE() - 90;
```

The Oracle version of the same query is shown here:

```
SELECT    P_CODE, P_INDATE, SYSDATE - 90 AS
          CUTDATE
FROM      PRODUCT
WHERE     P_INDATE <= SYSDATE - 90;
```

Note that DATE() and SYSDATE are special functions that return the current date in MS Access and Oracle, respectively. You can use the DATE() and SYSDATE functions anywhere a date literal is expected, such as in the value list of an INSERT statement, in an UPDATE statement when changing the value of a date attribute, or in a SELECT statement as shown here. Of course, the previous query output would change based on the current date.

Suppose that a manager wants a list of all products, the dates they were received, and the warranty expiration date (90 days from when the product was received). To generate that list, type:

```
SELECT    P_CODE, P_INDATE, P_INDATE + 90 AS EXPDATE
FROM      PRODUCT;
```

Note that you can use all arithmetic operators with date attributes as well as with numeric attributes.

### 7.4.2 Arithmetic Operators: The Rule of Precedence

As you saw in the previous example, you can use arithmetic operators with table attributes in a column list or in a conditional expression. In fact, SQL commands are often used in conjunction with the arithmetic operators shown in Table 7.7.

**TABLE 7.7 The Arithmetic Operators**

| ARITHMETIC OPERATOR | DESCRIPTION |
|---|---|
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |
| ^ | Raise to the power of (some applications use ** instead of ^) |

Do not confuse the multiplication symbol (*) with the wildcard symbol used by some SQL implementations, such as MS Access; the latter is used only in string comparisons, while the former is used in conjunction with mathematical procedures.

As you perform mathematical operations on attributes, remember the rules of precedence. As the name suggests, the **rules of precedence** are the rules that establish the order in which computations are completed. For example, note the order of the following computational sequence:

1. Perform operations within parentheses.
2. Perform power operations.
3. Perform multiplications and divisions.
4. Perform additions and subtractions.

The application of the rules of precedence will tell you that 8 + 2 * 5 = 8 + 10 = 18, but (8 + 2) * 5 = 10 * 5 = 50. Similarly, 4 + 5^2 * 3 = 4 + 25 * 3 = 79, but (4 + 5)^2 * 3 = 81 * 3 = 243, while the operation expressed by (4 + 5^2) * 3 yields the answer (4 + 25) * 3 = 29 * 3 = 87.

### 7.4.3 Logical Operators: AND, OR, and NOT

In the real world, a search of data normally involves multiple conditions. For example, when you are buying a new house, you look for a certain area, a certain number of bedrooms, bathrooms, stories, and so on. In the same way, SQL allows you to include multiple conditions in a query through the use of logical operators. The logical operators are AND, OR, and NOT. For example, if you want a list of the table contents for either the V_CODE = 21344 **or** the V_CODE = 24288, you can use the **OR** operator, as in the following command sequence:

```
SELECT    P_DESCRIPT, P_INDATE, P_PRICE, V_CODE
FROM      PRODUCT
WHERE     V_CODE = 21344 OR V_CODE = 24288;
```

That command generates the six rows shown in Figure 7.12 that match the logical restriction.

**FIGURE 7.12** Selected PRODUCT table attributes: the logical OR

| P_DESCRIPT | P_INDATE | P_PRICE | V_CODE |
|---|---|---|---|
| 7.25-in. pwr. saw blade | 13-Dec-09 | 14.99 | 21344 |
| 9.00-in. pwr. saw blade | 13-Nov-09 | 17.49 | 21344 |
| B&D jigsaw, 12-in. blade | 30-Dec-09 | 109.92 | 24288 |
| B&D jigsaw, 8-in. blade | 24-Dec-09 | 99.87 | 24288 |
| Rat-tail file, 1/8-in. fine | 15-Dec-09 | 4.99 | 21344 |
| Hicut chain saw, 16 in. | 07-Feb-10 | 256.99 | 24288 |

The logical **AND** has the same SQL syntax requirement. The following command generates a list of all rows for which P_PRICE is less than $50 and for which P_INDATE is a date occurring after January 15, 2010:

```
SELECT    P_DESCRIPT, P_INDATE, P_PRICE,
          V_CODE
FROM      PRODUCT
WHERE     P_PRICE < 50
AND       P_INDATE > '15-Jan-2010';
```

This command will produce the output shown in Figure 7.13.

**FIGURE 7.13** Selected PRODUCT table attributes: the logical AND

| P_DESCRIPT | P_INDATE | P_PRICE | V_CODE |
|---|---|---|---|
| B&D cordless drill, 1/2-in. | 20-Jan-10 | 38.95 | 25595 |
| Claw hammer | 20-Jan-10 | 9.95 | 21225 |
| PVC pipe, 3.5-in., 8-ft | 20-Feb-10 | 5.87 | |
| 1.25-in. metal screw, 25 | 01-Mar-10 | 6.99 | 21225 |
| 2.5-in. wd. screw, 50 | 24-Feb-10 | 8.45 | 21231 |

You can combine the logical OR with the logical AND to place further restrictions on the output. For example, suppose that you want a table listing for the following conditions:

- The P_INDATE is after January 15, 2010, and the P_PRICE is less than $50.
- Or the V_CODE is 24288.

The required listing can be produced by using:

```
SELECT    P_DESCRIPT, P_INDATE, P_PRICE, V_CODE
FROM      PRODUCT
WHERE     (P_PRICE < 50 AND P_INDATE > '15-Jan-2010')
OR        V_CODE = 24288;
```

Note the use of parentheses to combine logical restrictions. Where you place the parentheses depends on how you want the logical restrictions to be executed. Conditions listed within parentheses are always executed first. The preceding query yields the output shown in Figure 7.14.

**FIGURE 7.14** Selected PRODUCT table attributes: the logical AND and OR

| P_DESCRIPT | P_INDATE | P_PRICE | V_CODE |
|---|---|---|---|
| B&D jigsaw, 12-in. blade | 30-Dec-09 | 109.92 | 24288 |
| B&D jigsaw, 8-in. blade | 24-Dec-09 | 99.87 | 24288 |
| B&D cordless drill, 1/2-in. | 20-Jan-10 | 38.95 | 25595 |
| Claw hammer | 20-Jan-10 | 9.95 | 21225 |
| Hicut chain saw, 16 in. | 07-Feb-10 | 256.99 | 24288 |
| PVC pipe, 3.5-in., 8-ft | 20-Feb-10 | 5.87 | |
| 1.25-in. metal screw, 25 | 01-Mar-10 | 6.99 | 21225 |
| 2.5-in. wd. screw, 50 | 24-Feb-10 | 8.45 | 21231 |

Note that the three rows with the V_CODE = 24288 are included regardless of the P_INDATE and P_PRICE entries for those rows.

The use of the logical operators OR and AND can become quite complex when numerous restrictions are placed on the query. In fact, a specialty field in mathematics known as **Boolean algebra** is dedicated to the use of logical operators.

The logical operator **NOT** is used to negate the result of a conditional expression. That is, in SQL, all conditional expressions evaluate to true or false. If an expression is true,

the row is selected; if an expression is false, the row is not selected. The NOT logical operator is typically used to find the rows that *do not* match a certain condition. For example, if you want to see a listing of all rows for which the vendor code is not 21344, use the command sequence:

```
SELECT    *
FROM      PRODUCT
WHERE     NOT (V_CODE = 21344);
```

Note that the condition is enclosed in parentheses; that practice is optional, but it is highly recommended for clarity. The logical NOT can be combined with AND and OR.

**NOTE**

If your SQL version does not support the logical NOT, you can generate the required output by using the condition:

```
WHERE V_CODE <> 21344
```

If your version of SQL does not support <>, use:

```
WHERE V_CODE != 21344
```

### 7.4.4 Special Operators

ANSI-standard SQL allows the use of special operators in conjunction with the WHERE clause. These special operators include:

**BETWEEN**: Used to check whether an attribute value is within a range

**IS NULL**: Used to check whether an attribute value is null

**LIKE**: Used to check whether an attribute value matches a given string pattern

**IN**: Used to check whether an attribute value matches any value within a value list

**EXISTS**: Used to check whether a subquery returns any rows

#### The BETWEEN Special Operator

If you use software that implements a standard SQL, the operator BETWEEN may be used to check whether an attribute value is within a range of values. For example, if you want to see a listing for all products whose prices are between $50 and $100, use the following command sequence:

```
SELECT    *
FROM      PRODUCT
WHERE     P_PRICE BETWEEN 50.00 AND 100.00;
```

**NOTE**

NOTE TO ORACLE USERS
When using the BETWEEN special operator, always specify the lower range value first. If you list the higher range value first, Oracle will return an empty result set.

If your DBMS does not support BETWEEN, you can use:

```
SELECT    *
FROM      PRODUCT
WHERE     P_PRICE > 50.00 AND P_PRICE < 100.00;
```

## The IS NULL Special Operator

Standard SQL allows the use of IS NULL to check for a null attribute value. For example, suppose that you want to list all products that do not have a vendor assigned (V_CODE is null). Such a null entry could be found by using the command sequence:

```
SELECT    P_CODE, P_DESCRIPT, V_CODE
FROM      PRODUCT
WHERE     V_CODE IS NULL;
```

Similarly, if you want to check a null date entry, the command sequence is:

```
SELECT    P_CODE, P_DESCRIPT, P_INDATE
FROM      PRODUCT
WHERE     P_INDATE IS NULL;
```

Note that SQL uses a special operator to test for nulls. Why? Couldn't you just enter a condition such as "V_CODE = NULL"? No. Technically, NULL is not a "value" the way the number 0 (zero) or the blank space is, but instead a NULL is a special property of an attribute that represents precisely the absence of any value.

## The LIKE Special Operator

The LIKE special operator is used in conjunction with wildcards to find patterns within string attributes. Standard SQL allows you to use the percent sign (%) and underscore (_) wildcard characters to make matches when the entire string is not known:

- % means any and all *following* or preceding characters are eligible. For example,
  'J%' includes Johnson, Jones, Jernigan, July, and J-231Q.
  'Jo%' includes Johnson and Jones.
  '%n' includes Johnson and Jernigan.
- _ means any *one* character may be substituted for the underscore. For example,
  '_23-456-6789' includes 123-456-6789, 223-456-6789, and 323-456-6789.
  '_23-_56-678_' includes 123-156-6781, 123-256-6782, and 823-956-6788.
  '_o_es' includes Jones, Cones, Cokes, totes, and roles.

**NOTE**

Some RDBMSs, such as Microsoft Access, use the wildcard characters * and ? instead of % and _.

For example, the following query would find all VENDOR rows for contacts whose last names begin with *Smith*.

```
SELECT    V_NAME, V_CONTACT, V_AREACODE, V_PHONE
FROM      VENDOR
WHERE     V_CONTACT LIKE 'Smith%';
```

If you check the original VENDOR data in Figure 7.2 again, you'll see that this SQL query yields three records: two Smiths and one Smithson.

Keep in mind that most SQL implementations yield case-sensitive searches. For example, Oracle will not yield a result that includes *Jones* if you use the wildcard search delimiter 'jo%' in a search for last names. The reason is that *Jones* begins with a capital *J*, and your wildcard search starts with a lowercase *j*. On the other hand, MS Access searches are not case sensitive.

For example, suppose that you typed the following query in Oracle:

```
SELECT    V_NAME, V_CONTACT, V_AREACODE, V_PHONE
FROM      VENDOR
WHERE     V_CONTACT LIKE 'SMITH%';
```

No rows will be returned because character-based queries may be case sensitive. That is, an uppercase character has a different ASCII code than a lowercase character, causing *SMITH, Smith,* and *smith* to be evaluated as different (unequal) entries. Because the table contains no vendor whose last name begins with (uppercase) *SMITH,* the (uppercase) 'SMITH%' used in the query cannot be matched. Matches can be made only when the query entry is written exactly like the table entry.

Some RDBMSs, such as Microsoft Access, automatically make the necessary conversions to eliminate case sensitivity. Others, such as Oracle, provide a special UPPER function to convert both table and query character entries to uppercase. (The conversion is done in the computer's memory only; the conversion has no effect on how the value is actually stored in the table.) So if you want to avoid a no-match result based on case sensitivity, and if your RDBMS allows the use of the UPPER function, you can generate the same results by using the query:

```
SELECT    V_NAME, V_CONTACT, V_AREACODE, V_PHONE
FROM      VENDOR
WHERE     UPPER(V_CONTACT) LIKE 'SMITH%';
```

The preceding query produces a list that includes all rows containing a last name that begins with *Smith*, regardless of uppercase or lowercase letter combinations such as *Smith, smith,* and *SMITH*.

The logical operators may be used in conjunction with the special operators. For instance, the query:

```
SELECT    V_NAME, V_CONTACT, V_AREACODE, V_PHONE
FROM      VENDOR
WHERE     V_CONTACT NOT LIKE 'Smith%';
```

will yield an output of all vendors whose names do not start with *Smith.*

Suppose that you do not know whether a person's name is spelled Johnson or Johnsen. The wildcard character _ lets you find a match for either spelling. The proper search would be instituted by the query:

```
SELECT    *
FROM      VENDOR
WHERE     V_CONTACT LIKE 'Johns_n';
```

Thus, the wildcards allow you to make matches when only approximate spellings are known. Wildcard characters may be used in combinations. For example, the wildcard search based on the string '_l%' can yield the strings Al, Alton, Elgin, Blakeston, blank, bloated, and eligible.

## The IN Special Operator

Many queries that would require the use of the logical OR can be more easily handled with the help of the special operator IN. For example, the query:

```
SELECT      *
FROM        PRODUCT
WHERE       V_CODE = 21344
OR          V_CODE = 24288;
```

can be handled more efficiently with:

```
SELECT      *
FROM        PRODUCT
WHERE       V_CODE IN (21344, 24288);
```

Note that the IN operator uses a value list. All of the values in the list must be of the same data type. Each of the values in the value list is compared to the attribute—in this case, V_CODE. If the V_CODE value matches any of the values in the list, the row is selected. In this example, the rows selected will be only those in which the V_CODE is either 21344 or 24288.

If the attribute used is of a character data type, the list values must be enclosed in single quotation marks. For instance, if the V_CODE had been defined as CHAR(5) when the table was created, the preceding query would have read:

```
SELECT      *
FROM        PRODUCT
WHERE       V_CODE IN ('21344', '24288');
```

The IN operator is especially valuable when it is used in conjunction with subqueries. For example, suppose that you want to list the V_CODE and V_NAME of only those vendors who provide products. In that case, you could use a subquery within the IN operator to automatically generate the value list. The query would be:

```
SELECT      V_CODE, V_NAME
FROM        VENDOR
WHERE       V_CODE IN (SELECT V_CODE FROM PRODUCT);
```

The preceding query will be executed in two steps:

1. The inner query or subquery will generate a list of V_CODE values from the PRODUCT tables. Those V_CODE values represent the vendors who supply products.
2. The IN operator will compare the values generated by the subquery to the V_CODE values in the VENDOR table and will select only the rows with matching values—that is, the vendors who provide products.

The IN special operator will receive additional attention in Chapter 8, where you will learn more about subqueries.

## The EXISTS Special Operator

The EXISTS special operator can be used whenever there is a requirement to execute a command based on the result of another query. That is, if a subquery returns any rows, run the main query; otherwise, don't. For example, the following query will list all vendors, but only if there are products to order:

```
SELECT      *
FROM        VENDOR
WHERE       EXISTS (SELECT * FROM PRODUCT WHERE P_QOH <= P_MIN);
```

The EXISTS special operator is used in the following example to list all vendors, but only if there are products with the quantity on hand, less than double the minimum quantity:

```
SELECT      *
FROM        VENDOR
WHERE       EXISTS (SELECT * FROM PRODUCT WHERE P_QOH < P_MIN * 2);
```

The EXISTS special operator will receive additional attention in Chapter 8, where you will learn more about subqueries.

## 7.5 ADDITIONAL DATA DEFINITION COMMANDS

In this section, you will learn how to change (alter) table structures by changing attribute characteristics and by adding columns. Then you will learn how to do advanced data updates to the new columns. Finally, you will learn how to copy tables or parts of tables and how to delete tables.

All changes in the table structure are made by using the **ALTER TABLE** command, followed by a keyword that produces the specific change you want to make. Three options are available: ADD, MODIFY, and DROP. You use ADD to add a column, MODIFY to change column characteristics, and DROP to delete a column from a table. Most RDBMSs do not allow you to delete a column (unless the column does not contain any values) because such an action might delete crucial data that are used by other tables. The basic syntax to add or modify columns is:

ALTER TABLE *tablename*
    {ADD | MODIFY} ( *columnname datatype* [ {ADD | MODIFY} *columnname datatype*] ) ;

The ALTER TABLE command can also be used to add table constraints. In those cases, the syntax would be:

ALTER TABLE *tablename*
    ADD *constraint* [ ADD *constraint* ] ;

where *constraint* refers to a constraint definition similar to those you learned in Section 7.2.6.

You could also use the ALTER TABLE command to remove a column or table constraint. The syntax would be as follows:

ALTER TABLE *tablename*
    DROP{PRIMARY KEY | COLUMN *columnname* | CONSTRAINT *constraintname* };

Notice that when removing a constraint, you need to specify the name given to the constraint. That is one reason why you should always name your constraints in your CREATE TABLE or ALTER TABLE statement.

### 7.5.1 Changing a Column's Data Type

Using the ALTER syntax, the (integer) V_CODE in the PRODUCT table can be changed to a character V_CODE by using:

```
ALTER TABLE PRODUCT
    MODIFY (V_CODE CHAR(5));
```

*Some RDBMSs, such as Oracle, do not let you change data types unless the column to be changed is empty.* For example, if you want to change the V_CODE field from the current number definition to a character definition, the above command will yield an error message, because the V_CODE column already contains data. The error message is easily explained. Remember that the V_CODE in PRODUCT references the V_CODE in VENDOR. If you change the V_CODE data type, the data types don't match, and there is a referential integrity violation, which triggers the error message. If the V_CODE column does not contain data, the preceding command sequence will produce the

expected table structure alteration (if the foreign key reference was not specified during the creation of the PRODUCT table).

### 7.5.2 Changing a Column's Data Characteristics

If the column to be changed already contains data, you can make changes in the column's characteristics if those changes do not alter the data *type*. For example, if you want to increase the width of the P_PRICE column to nine digits, use the command:

```
ALTER TABLE PRODUCT
      MODIFY (P_PRICE DECIMAL(9,2));
```

If you now list the table contents, you can see that the column width of P_PRICE has increased by one digit.

**NOTE**

Some DBMSs impose limitations on when it's possible to change attribute characteristics. For example, Oracle lets you increase (but not decrease) the size of a column. The reason for this restriction is that an attribute modification will affect the integrity of the data in the database. In fact, some attribute changes can be done only when there are no data in any rows for the affected attribute.

### 7.5.3 Adding a Column

You can alter an existing table by adding one or more columns. In the following example, you add the column named P_SALECODE to the PRODUCT table. (This column will be used later to determine whether goods that have been in inventory for a certain length of time should be placed on special sale.)

Suppose that you expect the P_SALECODE entries to be 1, 2, or 3. Because there will be no arithmetic performed with the P_SALECODE, the P_SALECODE will be classified as a single-character attribute. Note the inclusion of all required information in the following ALTER command:

```
ALTER TABLE PRODUCT
      ADD (P_SALECODE CHAR(1));
```

**ONLINE CONTENT**

If you are using the MS Access databases provided in the Premium Website, you can track each of the updates in the following sections. For example, look at the copies of the PRODUCT table in the **Ch07_SaleCo** database, one named Product_2 and one named PRODUCT_3. Each of the two copies includes the new P_SALECODE column. If you want to see the *cumulative* effect of all UPDATE commands, you can continue using the PRODUCT table with the P_SALECODE modification and all of the changes you will make in the following sections. (You might even want to use both options, first to examine the individual effects of the update queries and then to examine the cumulative effects.)

When adding a column, be careful not to include the NOT NULL clause for the new column. Doing so will cause an error message; if you add a new column to a table that already has rows, the existing rows will default to a value of null for the new column. Therefore, it is not possible to add the NOT NULL clause for this new column. (You can, of course, add the NOT NULL clause to the table structure after all of the data for the new column have been entered and the column no longer contains nulls.)

### 7.5.4 Dropping a Column

Occasionally, you might want to modify a table by deleting a column. Suppose that you want to delete the V_ORDER attribute from the VENDOR table. To accomplish that, you would use the following command:

```
ALTER TABLE VENDOR
      DROP COLUMN V_ORDER;
```

Again, some RDBMSs impose restrictions on attribute deletion. For example, you may not drop attributes that are involved in foreign key relationships, nor may you delete an attribute of a table that contains only that one attribute.

### 7.5.5 Advanced Data Updates

To make data entries in an existing row's columns, SQL allows the UPDATE command. The UPDATE command updates only data in existing rows. For example, to enter the P_SALECODE value '2' in the fourth row, use the UPDATE command together with the primary key P_CODE '1546-QQ2'. Enter the value by using the command sequence:

```
UPDATE      PRODUCT
SET         P_SALECODE = '2'
WHERE       P_CODE = '1546-QQ2';
```

Subsequent data can be entered the same way, defining each entry location by its primary key (P_CODE) and its column location (P_SALECODE). For example, if you want to enter the P_SALECODE value '1' for the P_CODE values '2232/QWE' and '2232/QTY', you use:

```
UPDATE      PRODUCT
SET         P_SALECODE = '1'
WHERE       P_CODE IN ('2232/QWE', '2232/QTY');
```

If your RDBMS does not support IN, use the following command:

```
UPDATE      PRODUCT
SET         P_SALECODE = '1'
WHERE       P_CODE = '2232/QWE' OR P_CODE = '2232/QTY';
```

The results of your efforts can be checked by using:

```
SELECT      P_CODE, P_DESCRIPT, P_INDATE, P_PRICE, P_SALECODE
FROM        PRODUCT;
```

Although the UPDATE sequences just shown allow you to enter values into specified table cells, the process is very cumbersome. Fortunately, if a relationship can be established between the entries and the existing columns, the relationship can be used to assign values to their appropriate slots. For example, suppose that you want to place sales codes based on the P_INDATE into the table, using the following schedule:

| P_INDATE | P_SALECODE |
|---|---|
| before December 25, 2009 | 2 |
| between January 16, 2010, and February 10, 2010 | 1 |

Using the PRODUCT table, the following two command sequences make the appropriate assignments:

```
UPDATE      PRODUCT
SET         P_SALECODE = '2'
WHERE       P_INDATE < '25-Dec-2009';
```

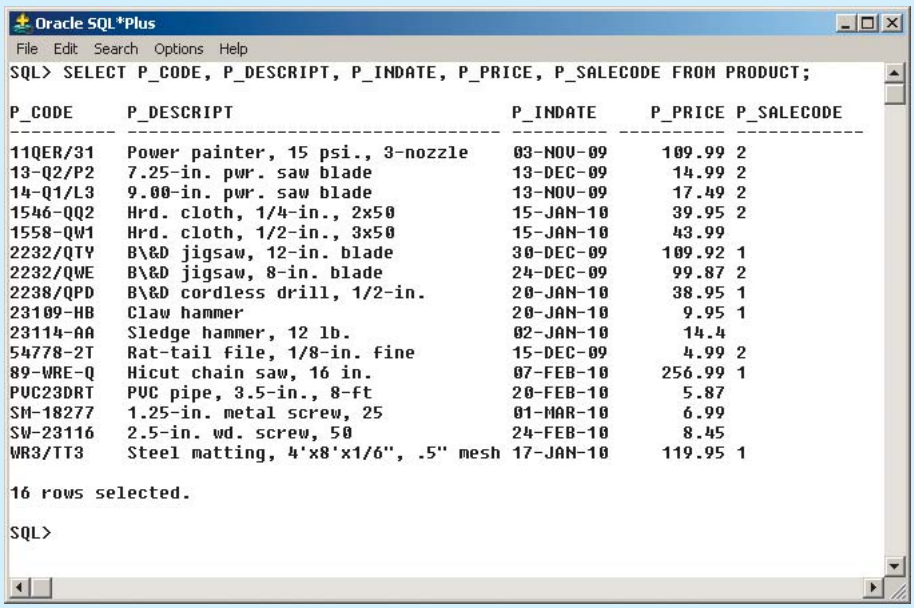
```
UPDATE      PRODUCT
SET         P_SALECODE = '1'
WHERE       P_INDATE >= '16-Jan-2010' AND P_INDATE <='10-Feb-2010';
```

To check the results of those two command sequences, use:

```
SELECT      P_CODE, P_DESCRIPT, P_INDATE, P_PRICE, P_SALECODE
FROM        PRODUCT;
```

If you have made *all* of the updates shown in this section using Oracle, your PRODUCT table should look like Figure 7.15. *Make sure that you issue a COMMIT statement to save these changes.*

**FIGURE 7.15** **The cumulative effect of the multiple updates in the PRODUCT table (Oracle)**




The arithmetic operators are particularly useful in data updates. For example, if the quantity on hand in your PRODUCT table has dropped below the minimum desirable value, you'll order more of the product. Suppose, for example, you have ordered 20 units of product 2232/QWE. When the 20 units arrive, you'll want to add them to inventory, using:

```
UPDATE      PRODUCT
SET         P_QOH = P_QOH + 20
WHERE       P_CODE = '2232/QWE';
```

If you want to add 10 percent to the price for all products that have current prices below $50, you can use:

```
UPDATE      PRODUCT
SET         P_PRICE = P_PRICE * 1.10
WHERE       P_PRICE < 50.00;
```

If you are using Oracle, issue a ROLLBACK command to undo the changes made by the last two UPDATE statements.

**NOTE**

If you fail to roll back the changes of the preceding UPDATE queries, the output of the subsequent queries will not match the results shown in the figures. Therefore:

- If you are using Oracle, use the ROLLBACK command to restore the database to its previous state.
- If you are using Access, copy the original **Ch07_SaleCo.mdb** file from the Premium Website for this book.

### 7.5.6 Copying Parts of Tables

As you will discover in later chapters on database design, sometimes it is necessary to break up a table structure into several component parts (or smaller tables). Fortunately, SQL allows you to copy the contents of selected table columns so that the data need not be reentered manually into the newly created table(s). For example, if you want to copy P_CODE, P_DESCRIPT, P_PRICE, and V_CODE from the PRODUCT table to a new table named PART, you create the PART table structure first, as follows:

```
CREATE TABLE PART(
PART_CODE          CHAR(8)          NOT NULL          UNIQUE,
PART_DESCRIPT      CHAR(35),
PART_PRICE         DECIMAL(8,2),
V_CODE             INTEGER,
PRIMARY KEY (PART_CODE));
```

Note that the PART column names need not be identical to those of the original table and that the new table need not have the same number of columns as the original table. In this case, the first column in the PART table is PART_CODE, rather than the original P_CODE found in the PRODUCT table. And the PART table contains only four columns rather than the seven columns found in the PRODUCT table. However, column characteristics must match; you cannot copy a character-based attribute into a numeric structure and vice versa.

Next, you need to add the rows to the new PART table, using the PRODUCT table rows. To do that, you use the INSERT command you learned in Section 7.3.7. The syntax is:

```
INSERT INTO    target_tablename[(target_columnlist)]
SELECT         source_columnlist
FROM           source_tablename;
```

Note that the target column list is required if the source column list doesn't match all of the attribute names and characteristics of the target table (including the order of the columns). Otherwise, you do not need to specify the target column list. In this example, you must specify the target column list in the INSERT command below because the column names of the target table are different:

```
INSERT INTO PART    (PART_CODE, PART_DESCRIPT, PART_PRICE, V_CODE)
SELECT              P_CODE, P_DESCRIPT, P_PRICE, V_CODE FROM PRODUCT;
```

The contents of the PART table can now be examined by using the query:

```
SELECT      * FROM PART;
```

to generate the new PART table's contents, shown in Figure 7.16.

**FIGURE 7.16** PART table attributes copied from the PRODUCT table

| PART_CODE | PART_DESCRIPT | PART_PRICE | V_CODE |
|---|---|---|---|
| 11QER/31 | Power painter, 15 psi., 3-nozzle | 109.99 | 25595 |
| 13-Q2/P2 | 7.25-in. pwr. saw blade | 14.99 | 21344 |
| 14-Q1/L3 | 9.00-in. pwr. saw blade | 17.49 | 21344 |
| 1546-QQ2 | Hrd. cloth, 1/4-in., 2x50 | 39.95 | 23119 |
| 1558-QW1 | Hrd. cloth, 1/2-in., 3x50 | 43.99 | 23119 |
| 2232/QTY | B&D jigsaw, 12-in. blade | 109.92 | 24288 |
| 2232/QWE | B&D jigsaw, 8-in. blade | 99.87 | 24288 |
| 2238/QPD | B&D cordless drill, 1/2-in. | 38.95 | 25595 |
| 23109-HB | Claw hammer | 9.95 | 21225 |
| 23114-AA | Sledge hammer, 12 lb. | 14.4 | |
| 54778-2T | Rat-tail file, 1/8-in. fine | 4.99 | 21344 |
| 89-WRE-Q | Hicut chain saw, 16 in. | 256.99 | 24288 |
| PVC23DRT | PVC pipe, 3.5-in., 8-ft | 5.87 | |
| SM-18277 | 1.25-in. metal screw, 25 | 6.99 | 21225 |
| SW-23116 | 2.5-in. wd. screw, 50 | 8.45 | 21231 |
| WR3/TT3 | Steel matting, 4'x8'x1/8", .5" mesh | 119.95 | 25595 |

SQL also provides another way to rapidly create a new table based on selected columns and rows of an existing table. In this case, the new table will copy the attribute names, data characteristics, and rows of the original table. The Oracle version of the command is:

```
CREATE TABLE PART AS
SELECT      P_CODE AS PART_CODE, P_DESCRIPT
            AS PART_DESCRIPT,
            P_PRICE AS PART_PRICE, V_CODE
FROM        PRODUCT;
```

If the PART table already exists, Oracle will not let you overwrite the existing table. To run this command, you must first delete the existing PART table. (See Section 7.5.8.)

The MS Access version of this command is:

```
SELECT      P_CODE AS PART_CODE, P_DESCRIPT AS PART_DESCRIPT,
            P_PRICE AS PART_PRICE, V_CODE INTO PART
FROM        PRODUCT;
```

If the PART table already exists, MS Access will ask if you want to delete the existing table and continue with the creation of the new PART table.

The SQL command just shown creates a new PART table with PART_CODE, PART_DESCRIPT, PART_PRICE, and V_CODE columns. In addition, all of the data rows (for the selected columns) will be copied automatically. *However, note that no entity integrity (primary key) or referential integrity (foreign key) rules are automatically applied to the new table.* In the next section, you will learn how to define the PK to enforce entity integrity and the FK to enforce referential integrity.

### 7.5.7 Adding Primary and Foreign Key Designations

When you create a new table based on another table, the new table does not include integrity rules from the old table. In particular, there is no primary key. To define the primary key for the new PART table, use the following command:

```
ALTER TABLE   PART
      ADD     PRIMARY KEY (PART_CODE);
```

Aside from the fact that the integrity rules are not automatically transferred to a new table that derives its data from one or more other tables, several other scenarios could leave you without entity and referential integrity. For example, you might have forgotten to define the primary and foreign keys when you created the original tables. Or if you imported tables from a different database, you might have discovered that the importing procedure did not transfer the integrity rules. In any case, you can reestablish the integrity rules by using the ALTER command. For example, if the PART table's foreign key has not yet been designated, it can be designated by:

```
ALTER TABLE   PART
      ADD     FOREIGN KEY (V_CODE) REFERENCES VENDOR;
```

Alternatively, if neither the PART table's primary key nor its foreign key has been designated, you can incorporate both changes at once, using:

```
ALTER TABLE   PART
      ADD     PRIMARY KEY (PART_CODE)
      ADD     FOREIGN KEY (V_CODE) REFERENCES VENDOR;
```

Even composite primary keys and multiple foreign keys can be designated in a single SQL command. For example, if you want to enforce the integrity rules for the LINE table shown in Figure 7.1, you can use:

```
ALTER TABLE   LINE
      ADD     PRIMARY KEY (INV_NUMBER, LINE_NUMBER)
      ADD     FOREIGN KEY (INV_NUMBER) REFERENCES INVOICE
      ADD     FOREIGN KEY (PROD_CODE) REFERENCES PRODUCT;
```

### 7.5.8 Deleting a Table from the Database

A table can be deleted from the database using the **DROP TABLE** command. For example, you can delete the PART table you just created with:

```
DROP TABLE PART;
```

You can drop a table only if that table is not the "one" side of any relationship. If you try to drop a table otherwise, the RDBMS will generate an error message indicating that a foreign key integrity violation has occurred.

## 7.6 ADDITIONAL SELECT QUERY KEYWORDS

One of the most important advantages of SQL is its ability to produce complex free-form queries. The logical operators that were introduced earlier to update table contents work just as well in the query environment. In addition, SQL provides useful functions that count, find minimum and maximum values, calculate averages, and so on. Better yet, SQL allows the user to limit queries to only those entries that have no duplicates or entries whose duplicates can be grouped.

### 7.6.1 Ordering a Listing

The **ORDER BY** clause is especially useful when the listing order is important to you. The syntax is:

```
SELECT      columnlist
FROM        tablelist
[WHERE      conditionlist ]
[ORDER BY   columnlist [ASC | DESC] ] ;
```

Although you have the option of declaring the order type—ascending or descending—the default order is ascending. For example, if you want the contents of the PRODUCT table listed by P_PRICE in ascending order, use:

```
SELECT      P_CODE, P_DESCRIPT, P_INDATE, P_PRICE
FROM        PRODUCT
ORDER BY    P_PRICE;
```

The output is shown in Figure 7.17. Note that ORDER BY yields an ascending price listing.

Comparing the listing in Figure 7.17 to the actual table contents shown earlier in Figure 7.2, you will see that in Figure 7.17, the lowest-priced product is listed first, followed by the next lowest-priced product, and so on. However, although ORDER BY produces a sorted output, the actual table contents are unaffected by the ORDER BY command.

To produce the list in descending order, you would enter:

```
SELECT      P_CODE, P_DESCRIPT, P_INDATE, P_PRICE
FROM        PRODUCT
ORDER BY    P_PRICE DESC;
```

# ADVANCED SQL

8

EIGHT

**In this chapter, you will learn:**

- About the relational set operators UNION, UNION ALL, INTERSECT, and MINUS
- How to use the advanced SQL JOIN operator syntax
- About the different types of subqueries and correlated queries
- How to use SQL functions to manipulate dates, strings, and other data
- How to create and use updatable views
- How to create and use triggers and stored procedures
- How to create embedded SQL

## Preview

In Chapter 7, Introduction to Structured Query Language (SQL), you learned the basic SQL data definition and data manipulation commands used to create and manipulate relational data. In this chapter, you build on what you learned in Chapter 7 and learn how to use more advanced SQL features.

In this chapter, you will learn about the SQL relational set operators (UNION, INTERSECT, and MINUS) and how those operators are used to merge the results of multiple queries. Joins are at the heart of SQL, so you must learn how to use the SQL JOIN statement to extract information from multiple tables. In the previous chapter, you learned how cascading queries inside other queries can be useful in certain circumstances. In this chapter, you will also learn about the different styles of subqueries that can be implemented in a SELECT statement. Finally, you will learn more of SQL's many functions to extract information from data, including manipulation of dates and strings and computations based on stored or even derived data.

In the real world, business procedures require the execution of clearly defined actions when a specific event occurs, such as the addition of a new invoice or a student's enrollment in a class. Such procedures can be applied within the DBMS through the use of triggers and stored procedures. In addition, SQL facilitates the application of business procedures when it is embedded in a programming language such as Visual Basic .NET, C#, or COBOL.

## ONLINE CONTENT

Although most of the examples used in this chapter are shown in Oracle, you could also use MS SQL Server. The Premium Website for this book provides you with the **ADVSQLDBINIT.SQL** script file (Oracle and MS SQL versions) to create the tables and load the data used in this chapter. There you will also find additional SQL script files to demonstrate each of the commands shown in this chapter.

## 8.1 RELATIONAL SET OPERATORS

In Chapter 3, The Relational Database Model, you learned about the eight general relational operators. In this section, you will learn how to use three SQL commands (UNION, INTERSECT, and MINUS) to implement the union, intersection, and difference relational operators.

In previous chapters, you learned that SQL data manipulation commands are **set-oriented**; that is, they operate over entire sets of rows and columns (tables) at once. Using sets, you can combine two or more sets to create new sets (or relations). That's precisely what the UNION, INTERSECT, and MINUS statements do. In relational database terms, you can use the words "sets," "relations," and "tables" interchangeably because they all provide a conceptual view of the data set as it is presented to the relational database user.

**NOTE**

The SQL standard defines the operations that all DBMSs must perform on data, but it leaves the implementation details to the DBMS vendors. Therefore, some advanced SQL features might not work on all DBMS implementations. Also, some DBMS vendors might implement additional features not found in the SQL standard.

UNION, INTERSECT, and MINUS are the names of the SQL statements implemented in Oracle. The SQL standard uses the keyword EXCEPT to refer to the difference (MINUS) relational operator. Other RDBMS vendors might use a different command name or might not implement a given command at all. To learn more about the ANSI/ISO SQL standards, check the ANSI Web site (*www.ansi.org*) to find out how to obtain the latest standard documents in electronic form. As of this writing, the most recent fully approved standard is SQL-2003. The SQL-2003 standard made revisions and additions to the previous standard; most notable is support for XML data. The SQL-2006 standard extended support for XML and multimedia data. The SQL-2008 standard added INSTEAD OF triggers and the TRUNCATE statement.

UNION, INTERSECT, and MINUS work properly only if relations are **union-compatible**, which means that the number of attributes must be the same and their corresponding data types must be alike. In practice, some RDBMS vendors require the data types to be "compatible" but not necessarily "exactly the same." For example, compatible data types are VARCHAR (35) and CHAR (15). In that case, both attributes store character (string) values; the only difference is the string size. Another example of compatible data types is NUMBER and SMALLINT. Both data types are used to store numeric values.

**NOTE**

Some DBMS products might require union-compatible tables to have *identical* data types.

## ONLINE CONTENT

The Premium Website for this book provides SQL script files (Oracle and MS SQL Server) to demonstrate the UNION, INTERSECT, and MINUS commands. It also provides the **Ch08_SaleCo** MS Access database containing supported set operator alternative queries.

### 8.1.1 UNION

Suppose SaleCo has bought another company. SaleCo's management wants to make sure that the acquired company's customer list is properly merged with SaleCo's customer list. Because it is quite possible that some customers have purchased goods from both companies, the two lists might contain common customers. SaleCo's management wants to make sure that customer records are not duplicated when the two customer lists are merged. The UNION query is a perfect tool for generating a combined listing of customers—one that excludes duplicate records.

The UNION statement combines rows from two or more queries *without including duplicate rows*. The syntax of the UNION statement is:

*query* UNION *query*

In other words, the UNION statement combines the output of two SELECT queries. (Remember that the SELECT statements must be union-compatible. That is, they must return the same number of attributes and similar data types.)

To demonstrate the use of the UNION statement in SQL, let's use the CUSTOMER and CUSTOMER_2 tables in the **Ch08_SaleCo** database. To show the combined CUSTOMER and CUSTOMER_2 records without the duplicates, the UNION query is written as follows:

```
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER
UNION
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER_2;
```

Figure 8.1 shows the contents of the CUSTOMER and CUSTOMER_2 tables and the result of the UNION query. Although MS Access is used to show the results here, similar results can be obtained with Oracle.

Note the following in Figure 8.1:

- The CUSTOMER table contains 10 rows, while the CUSTOMER_2 table contains 7 rows.
- Customers Dunne and Olowski are included in the CUSTOMER table as well as in the CUSTOMER_2 table.
- The UNION query yields 15 records because the duplicate records of customers Dunne and Olowski are not included. In short, the UNION query yields a unique set of records.

The UNION statement can be used to unite more than just two queries. For example, assume that you have four union-compatible queries named T1, T2, T3, and T4. With the UNION statement, you can combine the output of all four queries into a single result set. The SQL statement will be similar to this:

```
SELECT column-list FROM T1
UNION
SELECT column-list FROM T2
UNION
SELECT column-list FROM T3
UNION
SELECT column-list FROM T4;
```

**FIGURE 8.1** UNION query results

**Database name: CH08_SaleCo**

**Table name: CUSTOMER**

| CUS_CODE | CUS_LNAME | CUS_FNAME | CUS_INITIAL | CUS_AREACODE | CUS_PHONE | CUS_BALANCE |
|---|---|---|---|---|---|---|
| 10010 | Ramas | Alfred | A | 615 | 844-2573 | 0.00 |
| 10011 | Dunne | Leona | K | 713 | 894-1238 | 0.00 |
| 10012 | Smith | Kathy | W | 615 | 894-2285 | 345.86 |
| 10013 | Olowski | Paul | F | 615 | 894-2180 | 536.75 |
| 10014 | Orlando | Myron | | 615 | 222-1672 | 0.00 |
| 10015 | O'Brian | Amy | B | 713 | 442-3381 | 0.00 |
| 10016 | Brown | James | G | 615 | 297-1228 | 221.19 |
| 10017 | Williams | George | | 615 | 290-2556 | 768.93 |
| 10018 | Farriss | Anne | G | 713 | 382-7185 | 216.55 |
| 10019 | Smith | Olette | K | 615 | 297-3809 | 0.00 |

**Table name: CUSTOMER_2**

| CUS_CODE | CUS_LNAME | CUS_FNAME | CUS_INITIAL | CUS_AREACODE | CUS_PHONE |
|---|---|---|---|---|---|
| 345 | Terrell | Justine | H | 615 | 322-9870 |
| 347 | Olowski | Paul | F | 615 | 894-2180 |
| 351 | Hernandez | Carlos | J | 723 | 123-7654 |
| 352 | McDowell | George | | 723 | 123-7768 |
| 365 | Tirpin | Khaleed | G | 723 | 123-9876 |
| 368 | Lewis | Marie | J | 734 | 332-1789 |
| 369 | Dunne | Leona | K | 713 | 894-1238 |

**Query name: qryUNION-of-CUSTOMER-and-CUSTOMER_2**

| CUS_LNAME | CUS_FNAME | CUS_INITIAL | CUS_AREACODE | CUS_PHONE |
|---|---|---|---|---|
| Brown | James | G | 615 | 297-1228 |
| Dunne | Leona | K | 713 | 894-1238 |
| Farriss | Anne | G | 713 | 382-7185 |
| Hernandez | Carlos | J | 723 | 123-7654 |
| Lewis | Marie | J | 734 | 332-1789 |
| McDowell | George | | 723 | 123-7768 |
| O'Brian | Amy | B | 713 | 442-3381 |
| Olowski | Paul | F | 615 | 894-2180 |
| Orlando | Myron | | 615 | 222-1672 |
| Ramas | Alfred | A | 615 | 844-2573 |
| Smith | Kathy | W | 615 | 894-2285 |
| Smith | Olette | K | 615 | 297-3809 |
| Terrell | Justine | H | 615 | 322-9870 |
| Tirpin | Khaleed | G | 723 | 123-9876 |
| Williams | George | | 615 | 290-2556 |

**NOTE**

The SQL-2003 standard calls for the elimination of duplicate rows when the UNION SQL statement is used. However, some DBMS vendors might not adhere to that standard. Check your DBMS manual to see if the UNION statement is supported and if so, *how* it is supported.

### 8.1.2 UNION ALL

If SaleCo's management wants to know how many customers are on *both* the CUSTOMER and CUSTOMER_2 lists, a UNION ALL query can be used to produce a relation that retains the duplicate rows. Therefore, the following query will keep all rows from both queries (including the duplicate rows) and return 17 rows.

```
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER
UNION ALL
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER_2;
```

Running the preceding UNION ALL query produces the result shown in Figure 8.2.

Like the UNION statement, the UNION ALL statement can be used to unite more than just two queries.

### 8.1.3 INTERSECT

If SaleCo's management wants to know which customer records are duplicated in the CUSTOMER and CUSTOMER_2 tables, the INTERSECT statement can be used to combine rows from two queries, returning only the rows that appear in both sets. The syntax for the INTERSECT statement is:

*query* INTERSECT *query*

**FIGURE 8.2** UNION ALL query results

**Database name: CH08_SaleCo**

**Table name: CUSTOMER**

| CUS_CODE | CUS_LNAME | CUS_FNAME | CUS_INITIAL | CUS_AREACODE | CUS_PHONE | CUS_BALANCE |
|---|---|---|---|---|---|---|
| 10010 | Ramas | Alfred | A | 615 | 844-2573 | 0.00 |
| 10011 | Dunne | Leona | K | 713 | 894-1238 | 0.00 |
| 10012 | Smith | Kathy | W | 615 | 894-2285 | 345.86 |
| 10013 | Olowski | Paul | F | 615 | 894-2180 | 536.75 |
| 10014 | Orlando | Myron | | 615 | 222-1672 | 0.00 |
| 10015 | O'Brian | Amy | B | 713 | 442-3381 | 0.00 |
| 10016 | Brown | James | G | 615 | 297-1228 | 221.19 |
| 10017 | Williams | George | | 615 | 290-2556 | 768.93 |
| 10018 | Farriss | Anne | G | 713 | 382-7185 | 216.55 |
| 10019 | Smith | Olette | K | 615 | 297-3809 | 0.00 |

**Table name: CUSTOMER_2**

| CUS_CODE | CUS_LNAME | CUS_FNAME | CUS_INITIAL | CUS_AREACODE | CUS_PHONE |
|---|---|---|---|---|---|
| 345 | Terrell | Justine | H | 615 | 322-9870 |
| 347 | Olowski | Paul | F | 615 | 894-2180 |
| 351 | Hernandez | Carlos | J | 723 | 123-7654 |
| 352 | McDowell | George | | 723 | 123-7768 |
| 365 | Tirpin | Khaleed | G | 723 | 123-9876 |
| 368 | Lewis | Marie | J | 734 | 332-1789 |
| 369 | Dunne | Leona | K | 713 | 894-1238 |

**Query name: qryUNION-ALL-of-CUSTOMER-and-CUSTOMER_2**

| CUS_LNAME | CUS_FNAME | CUS_INITIAL | CUS_AREACODE | CUS_PHONE |
|---|---|---|---|---|
| Brown | James | G | 615 | 297-1228 |
| Dunne | Leona | K | 713 | 894-1238 |
| Farriss | Anne | G | 713 | 382-7185 |
| Hernandez | Carlos | J | 723 | 123-7654 |
| Lewis | Marie | J | 734 | 332-1789 |
| McDowell | George | | 723 | 123-7768 |
| O'Brian | Amy | B | 713 | 442-3381 |
| Olowski | Paul | F | 615 | 894-2180 |
| Orlando | Myron | | 615 | 222-1672 |
| Ramas | Alfred | A | 615 | 844-2573 |
| Smith | Kathy | W | 615 | 894-2285 |
| Smith | Olette | K | 615 | 297-3809 |
| Terrell | Justine | H | 615 | 322-9870 |
| Tirpin | Khaleed | G | 723 | 123-9876 |
| Williams | George | | 615 | 290-2556 |

To generate the list of duplicate customer records, you can use:

```
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER
INTERSECT
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER_2;
```

The INTERSECT statement can be used to generate additional useful customer information. For example, the following query returns the customer codes for all customers who are located in area code 615 and who have made purchases. (If a customer has made a purchase, there must be an invoice record for that customer.)

```
SELECT      CUS_CODE FROM CUSTOMER WHERE CUS_AREACODE = '615'
INTERSECT
SELECT      DISTINCT CUS_CODE FROM INVOICE;
```

Figure 8.3 shows both sets of SQL statements and their output.

### 8.1.4 MINUS

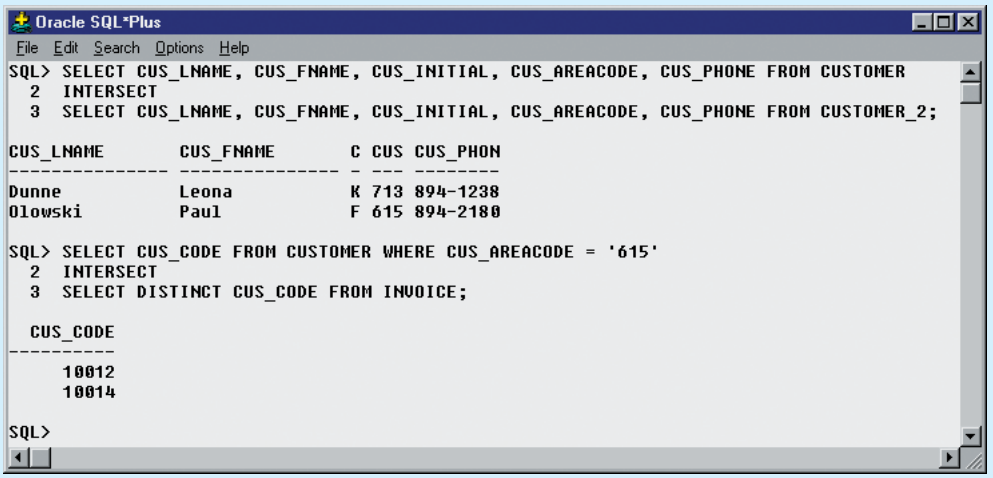
The MINUS statement in SQL combines rows from two queries and returns only the rows that appear in the first set but not in the second. The syntax for the MINUS statement is:

*query* MINUS *query*

For example, if the SaleCo managers want to know what customers in the CUSTOMER table are not found in the CUSTOMER_2 table, they can use:

```
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER
MINUS
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM        CUSTOMER_2;
```

**FIGURE 8.3** INTERSECT query results




**NOTE**

MS Access does not support the INTERSECT query, nor does it support other complex queries you will explore in this chapter. At least in some cases, Access might be able to give you the desired results if you use an alternative query format or procedure. For example, although Access does not support SQL triggers and stored procedures, you can use Visual Basic code to perform similar actions. However, the objective here is to show you how some important standard SQL features may be used.

If the managers want to know what customers in the CUSTOMER_2 table are not found in the CUSTOMER table, they merely switch the table designations:

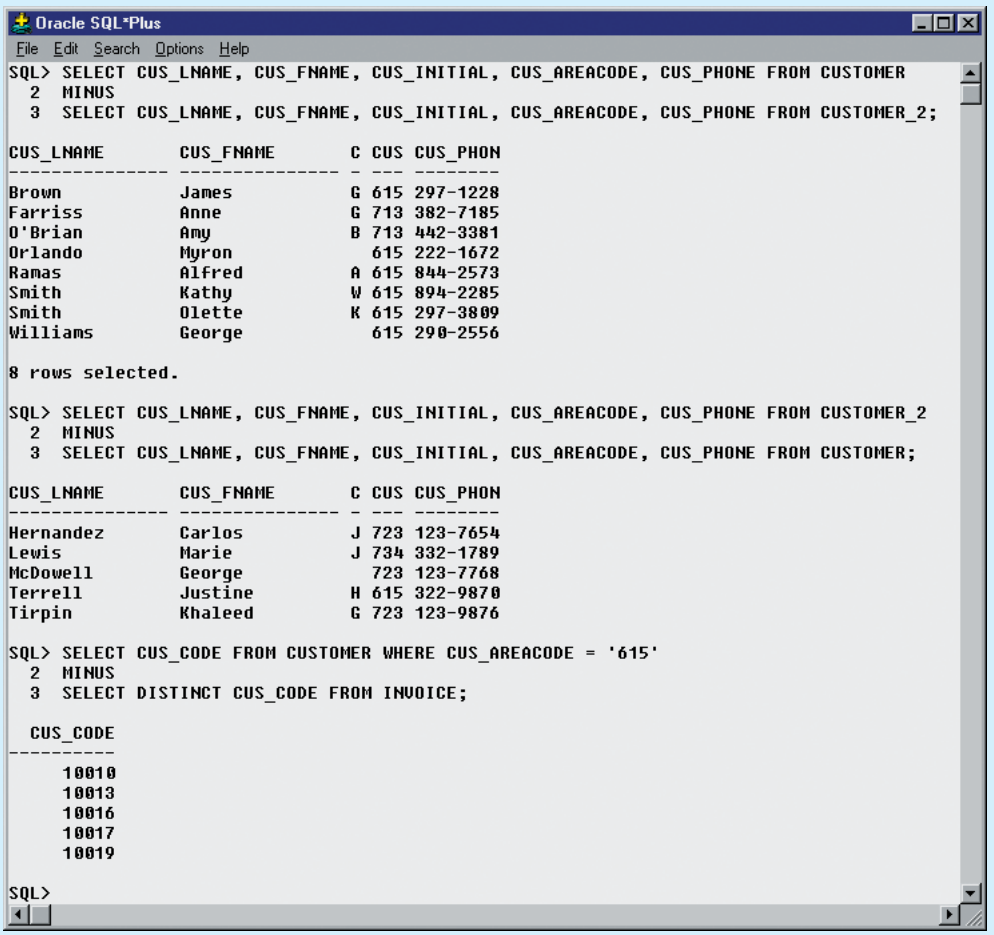
```
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER_2
MINUS
SELECT    CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE, CUS_PHONE
FROM      CUSTOMER;
```

You can extract much useful information by combining MINUS with various clauses such as WHERE. For example, the following query returns the customer codes for all customers located in area code 615 minus the ones who have made purchases, leaving the customers in area code 615 who have not made purchases.

```
SELECT    CUS_CODE FROM CUSTOMER WHERE CUS_AREACODE = '615'
MINUS
SELECT    DISTINCT CUS_CODE FROM INVOICE;
```

Figure 8.4 shows the preceding three SQL statements and their output.

**FIGURE 8.4** MINUS query results




**NOTE**

Some DBMS products do not support the INTERSECT or MINUS statements, while others might implement the difference relational operator in SQL as EXCEPT. Consult your DBMS manual to see if the statements illustrated here are supported by your DBMS.

### 8.1.5 Syntax Alternatives

If your DBMS doesn't support the INTERSECT or MINUS statements, you can use the IN and NOT IN subqueries to obtain similar results. For example, the following query will produce the same results as the INTERSECT query shown in Section 8.1.3:

```
SELECT    CUS_CODE FROM CUSTOMER
WHERE     CUS_AREACODE = '615' AND
          CUS_CODE IN (SELECT DISTINCT CUS_CODE FROM INVOICE);
```

Figure 8.5 shows the use of the INTERSECT alternative.

**FIGURE 8.5 INTERSECT alternative**

**Database name: CH08_SaleCo**

**Table name: CUSTOMER**

| CUS_CODE | CUS_LNAME | CUS_FNAME | CUS_INITIAL | CUS_AREACODE | CUS_PHONE | CUS_BALANCE |
|---|---|---|---|---|---|---|
| 10010 | Ramas | Alfred | A | 615 | 844-2573 | 0.00 |
| 10011 | Dunne | Leona | K | 713 | 894-1238 | 0.00 |
| 10012 | Smith | Kathy | W | 615 | 894-2285 | 345.86 |
| 10013 | Olowski | Paul | F | 615 | 894-2180 | 536.75 |
| 10014 | Orlando | Myron | | 615 | 222-1672 | 0.00 |
| 10015 | O'Brian | Amy | B | 713 | 442-3381 | 0.00 |
| 10016 | Brown | James | G | 615 | 297-1228 | 221.19 |
| 10017 | Williams | George | | 615 | 290-2556 | 768.93 |
| 10018 | Farriss | Anne | G | 713 | 382-7185 | 216.55 |
| 10019 | Smith | Olette | K | 615 | 297-3809 | 0.00 |

**Table name: INVOICE**

| INV_NUMBER | CUS_CODE | INV_DATE |
|---|---|---|
| 1001 | 10014 | 16-Jan-10 |
| 1002 | 10011 | 16-Jan-10 |
| 1003 | 10012 | 16-Jan-10 |
| 1004 | 10011 | 17-Jan-10 |
| 1005 | 10018 | 17-Jan-10 |
| 1006 | 10014 | 17-Jan-10 |
| 1007 | 10015 | 17-Jan-10 |
| 1008 | 10011 | 17-Jan-10 |

**Query name: qry-INTERSECT-Alternative**

| CUS_CODE |
|---|
| 10012 |
| 10014 |

**NOTE**

MS Access will generate an input request for the CUS_AREACODE if you use apostrophes around the area code. (If you supply the 615 area code, the query will execute properly.) You can eliminate that problem by using standard double quotation marks, writing the WHERE clause in the second line of the preceding SQL statement as:

WHERE CUS_AREACODE = "615" AND

MS Access will also accept single quotation marks.

Using the same alternative to the MINUS statement, you can generate the output for the third MINUS query shown in Section 8.1.4 by using:

```
SELECT    CUS_CODE FROM CUSTOMER
WHERE     CUS_AREACODE = '615' AND
          CUS_CODE NOT IN (SELECT DISTINCT CUS_CODE FROM INVOICE);
```

The results of that query are shown in Figure 8.6. Note that the query output includes only the customers in area code 615 who have not made any purchases and, therefore, have not generated invoices.

**FIGURE 8.6** MINUS alternative

**Database name: CH08_SaleCo**

**Table name: CUSTOMER**

| CUS_CODE | CUS_LNAME | CUS_FNAME | CUS_INITIAL | CUS_AREACODE | CUS_PHONE | CUS_BALANCE |
|---|---|---|---|---|---|---|
| 10010 | Ramas | Alfred | A | 615 | 844-2573 | 0.00 |
| 10011 | Dunne | Leona | K | 713 | 894-1238 | 0.00 |
| 10012 | Smith | Kathy | W | 615 | 894-2285 | 345.86 |
| 10013 | Olowski | Paul | F | 615 | 894-2180 | 536.75 |
| 10014 | Orlando | Myron | | 615 | 222-1672 | 0.00 |
| 10015 | O'Brian | Amy | B | 713 | 442-3381 | 0.00 |
| 10016 | Brown | James | G | 615 | 297-1228 | 221.19 |
| 10017 | Williams | George | | 615 | 290-2556 | 768.93 |
| 10018 | Farriss | Anne | G | 713 | 382-7185 | 216.55 |
| 10019 | Smith | Olette | K | 615 | 297-3809 | 0.00 |

**Table name: INVOICE**

| INV_NUMBER | CUS_CODE | INV_DATE |
|---|---|---|
| 1001 | 10014 | 16-Jan-10 |
| 1002 | 10011 | 16-Jan-10 |
| 1003 | 10012 | 16-Jan-10 |
| 1004 | 10011 | 17-Jan-10 |
| 1005 | 10018 | 17-Jan-10 |
| 1006 | 10014 | 17-Jan-10 |
| 1007 | 10015 | 17-Jan-10 |
| 1008 | 10011 | 17-Jan-10 |

**Query name: qry-MINUS-Alternative**

| CUS_CODE |
|---|
| 10010 |
| 10013 |
| 10016 |
| 10017 |
| 10019 |

## 8.2 SQL JOIN OPERATORS

The relational join operation merges rows from two tables and returns the rows with one of the following conditions:

- Have common values in common columns (natural join).
- Meet a given join condition (equality or inequality).
- Have common values in common columns or have no matching values (outer join).

In Chapter 7, you learned how to use the SELECT statement in conjunction with the WHERE clause to join two or more tables. For example, you can join the PRODUCT and VENDOR tables through their common V_CODE by writing:

```
SELECT    P_CODE, P_DESCRIPT, P_PRICE, V_NAME
FROM      PRODUCT, VENDOR
WHERE     PRODUCT.V_CODE = VENDOR.V_CODE;
```

The preceding SQL join syntax is sometimes referred to as an "old-style" join. Note that the FROM clause contains the tables being joined and that the WHERE clause contains the condition(s) used to join the tables.

Note the following points about the preceding query:

- The FROM clause indicates which tables are to be joined. If three or more tables are included, the join operation takes place two tables at a time, from left to right. For example, if you are joining tables T1, T2, and T3, the first join is table T1 with T2; the results of that join are then joined to table T3.
- The join condition in the WHERE clause tells the SELECT statement which rows will be returned. In this case, the SELECT statement returns all rows for which the V_CODE values in the PRODUCT and VENDOR tables are equal.
- The number of join conditions is always equal to the number of tables being joined minus one. For example, if you join three tables (T1, T2, and T3), you will have two join conditions (j1 and j2). All join conditions are connected through an AND logical operator. The first join condition (j1) defines the join criteria for T1 and T2. The second join condition (j2) defines the join criteria for the output of the first join and T3.

- Generally, the join condition will be an equality comparison of the primary key in one table and the related foreign key in the second table.

Join operations can be classified as inner joins and outer joins. The **inner join** is the traditional join in which only rows that meet a given criteria are selected. The join criteria can be an equality condition (also called a natural join or an equijoin) or an inequality condition (also called a theta join). An **outer join** returns not only the matching rows but also the rows with unmatched attribute values for one table or both tables to be joined. The SQL standard also introduces a special type of join, called a **cross join**, that returns the same result as the Cartesian product of two sets or tables.

In this section, you will learn various ways to express join operations that meet the ANSI SQL standard. These are outlined in Table 8.1. It is useful to remember that not all DBMS vendors provide the same level of SQL support and that some do not support the join styles shown in this section. Oracle 11g is used to demonstrate the use of the following queries. Refer to your DBMS manual if you are using a different DBMS.

**TABLE 8.1 SQL Join Expression Styles**

| JOIN CLASSIFICATION | JOIN TYPE | SQL SYNTAX EXAMPLE | DESCRIPTION |
|---|---|---|---|
| CROSS | CROSS JOIN | SELECT *<br>FROM T1, T2 | Returns the Cartesian product of T1 and T2 (old style). |
| | | SELECT *<br>FROM T1 CROSS JOIN T2 | Returns the Cartesian product of T1 and T2. |
| INNER | Old-Style JOIN | SELECT *<br>FROM T1, T2<br>WHERE T1.C1=T2.C1 | Returns only the rows that meet the join condition in the WHERE clause (old style). Only rows with matching values are selected. |
| | NATURAL JOIN | SELECT *<br>FROM T1 NATURAL JOIN T2 | Returns only the rows with matching values in the matching columns. The matching columns must have the same names and similar data types. |
| | JOIN USING | SELECT *<br>FROM T1 JOIN T2 USING (C1) | Returns only the rows with matching values in the columns indicated in the USING clause. |
| | JOIN ON | SELECT *<br>FROM T1 JOIN T2<br>ON T1.C1=T2.C1 | Returns only the rows that meet the join condition indicated in the ON clause. |
| OUTER | LEFT JOIN | SELECT *<br>FROM T1 LEFT OUTER JOIN T2<br>ON T1.C1=T2.C1 | Returns rows with matching values and includes all rows from the left table (T1) with unmatched values. |
| | RIGHT JOIN | SELECT *<br>FROM T1 RIGHT OUTER JOIN T2<br>ON T1.C1=T2.C1 | Returns rows with matching values and includes all rows from the right table (T2) with unmatched values. |
| | FULL JOIN | SELECT *<br>FROM T1 FULL OUTER JOIN T2<br>ON T1.C1=T2.C1 | Returns rows with matching values and includes all rows from both tables (T1 and T2) with unmatched values. |

### 8.2.1 Cross Join

A **cross join** performs a relational product (also known as the Cartesian product) of two tables. The cross join syntax is:

SELECT *column-list* FROM *table1* CROSS JOIN *table2*

For example,

```
SELECT * FROM INVOICE CROSS JOIN LINE;
```

performs a cross join of the INVOICE and LINE tables. That CROSS JOIN query generates 144 rows. (There were 8 invoice rows and 18 line rows, yielding 8 × 18 = 144 rows.)

You can also perform a cross join that yields only specified attributes. For example, you can specify:

```
SELECT    INVOICE.INV_NUMBER, CUS_CODE, INV_DATE, P_CODE
FROM      INVOICE CROSS JOIN LINE;
```

The results generated through that SQL statement can also be generated by using the following syntax:

```
SELECT    INVOICE.INV_NUMBER, CUS_CODE, INV_DATE, P_CODE
FROM      INVOICE, LINE;
```

### 8.2.2 Natural Join

Recall from Chapter 3 that a natural join returns all rows with matching values in the matching columns and eliminates duplicate columns. That style of query is used when the tables share one or more common attributes with common names. The natural join syntax is:

SELECT *column-list* FROM *table1* NATURAL JOIN *table2*

The natural join will perform the following tasks:

- Determine the common attribute(s) by looking for attributes with identical names and compatible data types.
- Select only the rows with common values in the common attribute(s).
- If there are no common attributes, return the relational product of the two tables.

The following example performs a natural join of the CUSTOMER and INVOICE tables and returns only selected attributes:

```
SELECT    CUS_CODE, CUS_LNAME, INV_NUMBER, INV_DATE
FROM      CUSTOMER NATURAL JOIN INVOICE;
```

The SQL code and its results are shown at the top of Figure 8.7.

You are not limited to two tables when performing a natural join. For example, you can perform a natural join of the INVOICE, LINE, and PRODUCT tables and project only selected attributes by writing:

```
SELECT    INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
FROM      INVOICE NATURAL JOIN LINE NATURAL JOIN PRODUCT;
```

The SQL code and its results are shown at the bottom of Figure 8.7.

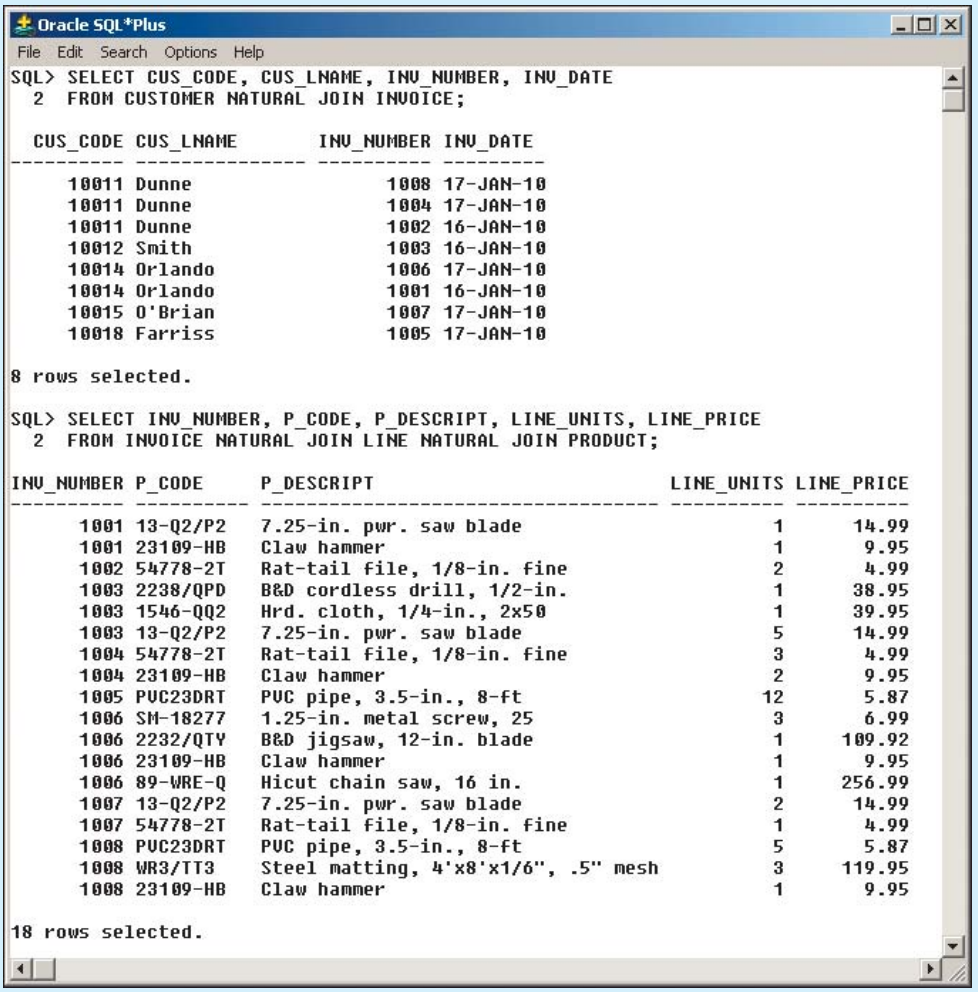
One important difference between the natural join and the "old-style" join syntax is that the natural join does not require the use of a table qualifier for the common attributes. In the first natural join example, you projected CUS_CODE. However, the projection did not require any table qualifier, even though the CUS_CODE attribute appeared in both CUSTOMER and INVOICE tables. The same can be said of the INV_NUMBER attribute in the second natural join example.

### 8.2.3 Join USING Clause

A second way to express a join is through the USING keyword. That query returns only the rows with matching values in the column indicated in the USING clause—and that column must exist in both tables. The syntax is:

SELECT *column-list* FROM *table1* JOIN *table2* USING (*common-column*)

**FIGURE 8.7** NATURAL JOIN results



```
Oracle SQL*Plus
File  Edit  Search  Options  Help
SQL> SELECT CUS_CODE, CUS_LNAME, INV_NUMBER, INV_DATE
  2  FROM CUSTOMER NATURAL JOIN INVOICE;

  CUS_CODE CUS_LNAME       INV_NUMBER INV_DATE
---------- --------------- ---------- ---------
     10011 Dunne                 1008 17-JAN-10
     10011 Dunne                 1004 17-JAN-10
     10011 Dunne                 1002 16-JAN-10
     10012 Smith                 1003 16-JAN-10
     10014 Orlando               1006 17-JAN-10
     10014 Orlando               1001 16-JAN-10
     10015 O'Brian               1007 17-JAN-10
     10018 Farriss               1005 17-JAN-10

8 rows selected.

SQL> SELECT INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
  2  FROM INVOICE NATURAL JOIN LINE NATURAL JOIN PRODUCT;

INV_NUMBER P_CODE     P_DESCRIPT                         LINE_UNITS LINE_PRICE
---------- ---------- ---------------------------------- ---------- ----------
      1001 13-Q2/P2   7.25-in. pwr. saw blade                    1      14.99
      1001 23109-HB   Claw hammer                               1       9.95
      1002 54778-2T   Rat-tail file, 1/8-in. fine               2       4.99
      1003 2238/QPD   B&D cordless drill, 1/2-in.               1      38.95
      1003 1546-QQ2   Hrd. cloth, 1/4-in., 2x50                 1      39.95
      1003 13-Q2/P2   7.25-in. pwr. saw blade                    5      14.99
      1004 54778-2T   Rat-tail file, 1/8-in. fine               3       4.99
      1004 23109-HB   Claw hammer                               2       9.95
      1005 PVC23DRT   PVC pipe, 3.5-in., 8-ft                  12       5.87
      1006 SM-18277   1.25-in. metal screw, 25                   3       6.99
      1006 2232/QTY   B&D jigsaw, 12-in. blade                   1     109.92
      1006 23109-HB   Claw hammer                               1       9.95
      1006 89-WRE-Q   Hicut chain saw, 16 in.                    1     256.99
      1007 13-Q2/P2   7.25-in. pwr. saw blade                    2      14.99
      1007 54778-2T   Rat-tail file, 1/8-in. fine               1       4.99
      1008 PVC23DRT   PVC pipe, 3.5-in., 8-ft                   5       5.87
      1008 WR3/TT3    Steel matting, 4'x8'x1/6", .5" mesh       3     119.95
      1008 23109-HB   Claw hammer                               1       9.95

18 rows selected.
```

To see the JOIN USING query in action, let's perform a join of the INVOICE and LINE tables by writing:

```
SELECT    INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
FROM      INVOICE JOIN LINE USING (INV_NUMBER) JOIN PRODUCT USING (P_CODE);
```

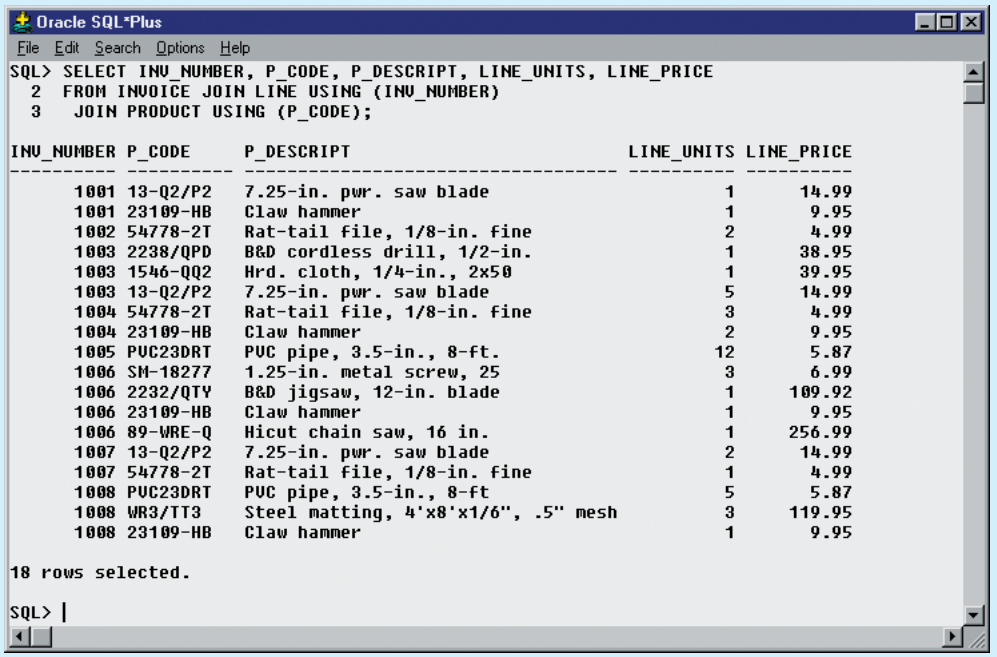
The SQL statement produces the results shown in Figure 8.8.

As was the case with the NATURAL JOIN command, the JOIN USING operand does not require table qualifiers. As a matter of fact, Oracle will return an error if you specify the table name in the USING clause.

### 8.2.4 JOIN ON Clause

The previous two join styles used common attribute names in the joining tables. Another way to express a join when the tables have no common attribute names is to use the JOIN ON operand. That query will return only the rows that meet the indicated join condition. The join condition will typically include an equality comparison expression of two

**FIGURE 8.8** JOIN USING results



```
Oracle SQL*Plus
File  Edit  Search  Options  Help
SQL> SELECT INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
  2  FROM INVOICE JOIN LINE USING (INV_NUMBER)
  3   JOIN PRODUCT USING (P_CODE);

INV_NUMBER P_CODE     P_DESCRIPT                         LINE_UNITS LINE_PRICE
---------- ---------- ---------------------------------- ---------- ----------
      1001 13-Q2/P2   7.25-in. pwr. saw blade                    1      14.99
      1001 23109-HB   Claw hammer                               1       9.95
      1002 54778-2T   Rat-tail file, 1/8-in. fine               2       4.99
      1003 2238/QPD   B&D cordless drill, 1/2-in.               1      38.95
      1003 1546-QQ2   Hrd. cloth, 1/4-in., 2x50                 1      39.95
      1003 13-Q2/P2   7.25-in. pwr. saw blade                    5      14.99
      1004 54778-2T   Rat-tail file, 1/8-in. fine               3       4.99
      1004 23109-HB   Claw hammer                               2       9.95
      1005 PVC23DRT   PVC pipe, 3.5-in., 8-ft.                 12       5.87
      1006 SM-18277   1.25-in. metal screw, 25                   3       6.99
      1006 2232/QTY   B&D jigsaw, 12-in. blade                   1     109.92
      1006 23109-HB   Claw hammer                               1       9.95
      1006 89-WRE-Q   Hicut chain saw, 16 in.                    1     256.99
      1007 13-Q2/P2   7.25-in. pwr. saw blade                    2      14.99
      1007 54778-2T   Rat-tail file, 1/8-in. fine               1       4.99
      1008 PVC23DRT   PVC pipe, 3.5-in., 8-ft                   5       5.87
      1008 WR3/TT3    Steel matting, 4'x8'x1/6", .5" mesh       3     119.95
      1008 23109-HB   Claw hammer                               1       9.95

18 rows selected.

SQL>
```

columns. (The columns may or may not share the same name but, obviously, must have comparable data types.) The syntax is:

SELECT *column-list* FROM *table1* JOIN *table2* ON *join-condition*

The following example performs a join of the INVOICE and LINE tables, using the ON clause. The result is shown in Figure 8.9.

```
SELECT    INVOICE.INV_NUMBER, PRODUCT.P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
FROM      INVOICE JOIN LINE ON INVOICE.INV_NUMBER = LINE.INV_NUMBER
          JOIN PRODUCT ON LINE.P_CODE = PRODUCT.P_CODE;
```

Note that unlike the NATURAL JOIN and the JOIN USING operands, the JOIN ON clause requires a table qualifier for the common attributes. If you do not specify the table qualifier, you will get a "column ambiguously defined" error message.

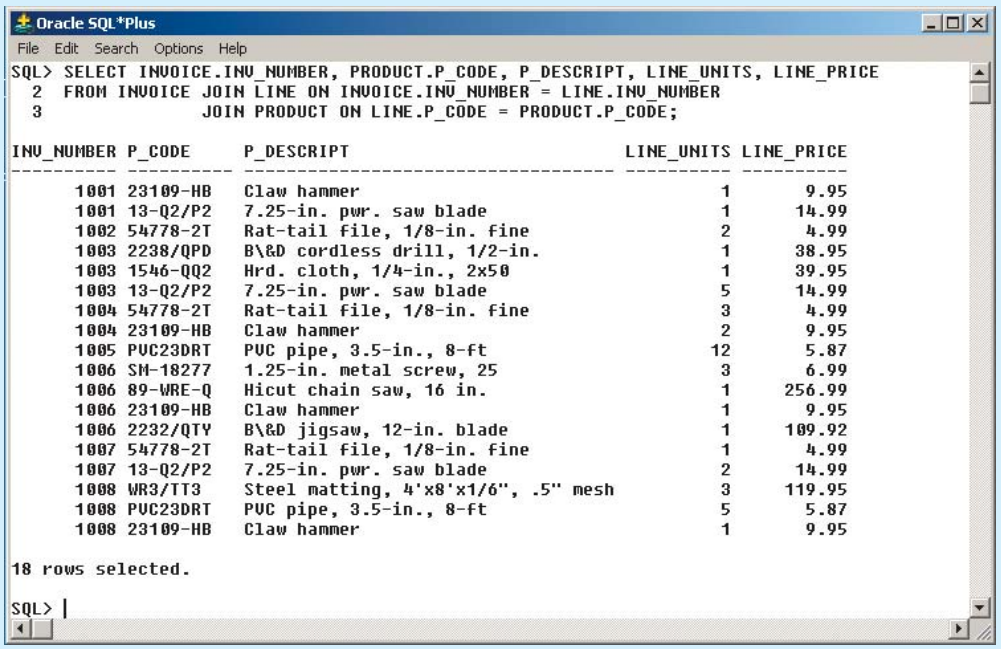
Keep in mind that the JOIN ON syntax lets you perform a join even when the tables do not share a common attribute name. For example, to generate a list of all employees with the managers' names, you can use the following (recursive) query:

```
SELECT    E.EMP_MGR, M.EMP_LNAME, E.EMP_NUM, E.EMP_LNAME
FROM      EMP E JOIN EMP M ON E.EMP_MGR = M.EMP_NUM
ORDER BY  E.EMP_MGR;
```

### 8.2.5 Outer Joins

An outer join returns not only the rows matching the join condition (that is, rows with matching values in the common columns) but also the rows with unmatched values. The ANSI standard defines three types of outer joins: left, right, and full. The left and right designations reflect the order in which the tables are processed by the DBMS. Remember that join operations take place two tables at a time. The first table named in the FROM clause will be the left side, and

**FIGURE 8.9** JOIN ON results



```
Oracle SQL*Plus
File  Edit  Search  Options  Help
SQL> SELECT INVOICE.INV_NUMBER, PRODUCT.P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE
  2  FROM INVOICE JOIN LINE ON INVOICE.INV_NUMBER = LINE.INV_NUMBER
  3               JOIN PRODUCT ON LINE.P_CODE = PRODUCT.P_CODE;

INV_NUMBER P_CODE     P_DESCRIPT                          LINE_UNITS LINE_PRICE
---------- ---------- ----------------------------------- ---------- ----------
      1001 23109-HB   Claw hammer                                  1       9.95
      1001 13-Q2/P2   7.25-in. pwr. saw blade                      1      14.99
      1002 54778-2T   Rat-tail file, 1/8-in. fine                  2       4.99
      1003 2238/QPD   B\&D cordless drill, 1/2-in.                 1      38.95
      1003 1546-QQ2   Hrd. cloth, 1/4-in., 2x50                    1      39.95
      1003 13-Q2/P2   7.25-in. pwr. saw blade                      5      14.99
      1004 54778-2T   Rat-tail file, 1/8-in. fine                  3       4.99
      1004 23109-HB   Claw hammer                                  2       9.95
      1005 PVC23DRT   PVC pipe, 3.5-in., 8-ft                     12       5.87
      1006 SM-18277   1.25-in. metal screw, 25                     3       6.99
      1006 89-WRE-Q   Hicut chain saw, 16 in.                      1     256.99
      1006 23109-HB   Claw hammer                                  1       9.95
      1006 2232/QTY   B\&D jigsaw, 12-in. blade                    1     109.92
      1007 54778-2T   Rat-tail file, 1/8-in. fine                  1       4.99
      1007 13-Q2/P2   7.25-in. pwr. saw blade                      2      14.99
      1008 WR3/TT3    Steel matting, 4'x8'x1/6", .5" mesh          3     119.95
      1008 PVC23DRT   PVC pipe, 3.5-in., 8-ft                      5       5.87
      1008 23109-HB   Claw hammer                                  1       9.95

18 rows selected.

SQL>
```

the second table named will be the right side. If three or more tables are being joined, the result of joining the first two tables becomes the left side, and the third table becomes the right side.

The left outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column) but also the rows in the left side table with unmatched values in the right side table. The syntax is:

SELECT column-list
FROM *table1* LEFT [OUTER] JOIN *table2* ON *join-condition*

For example, the following query lists the product code, vendor code, and vendor name for all products and includes those vendors with no matching products:

```
SELECT    P_CODE, VENDOR.V_CODE, V_NAME
FROM      VENDOR LEFT JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;
```

The preceding SQL code and its results are shown in Figure 8.10.

The right outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column) but also the rows in the right side table with unmatched values in the left side table. The syntax is:

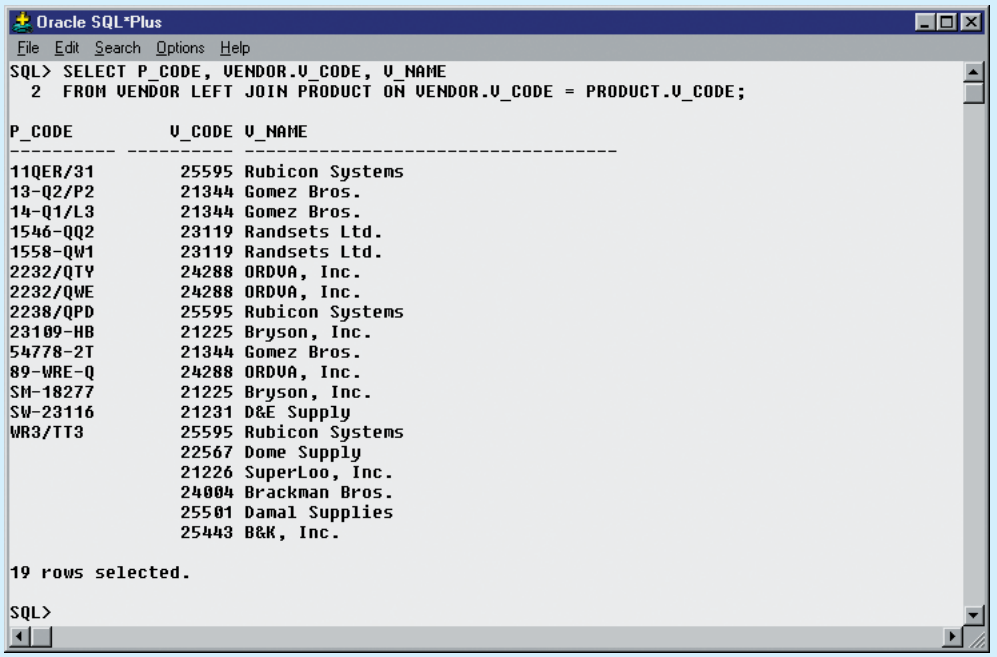
SELECT column-list
FROM *table1* RIGHT [OUTER] JOIN *table2* ON *join-condition*

For example, the following query lists the product code, vendor code, and vendor name for all products and also includes those products that do not have a matching vendor code:

```
SELECT    P_CODE, VENDOR.V_CODE, V_NAME
FROM      VENDOR RIGHT JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;
```

The SQL code and its output are shown in Figure 8.11.

**FIGURE 8.10** LEFT JOIN results



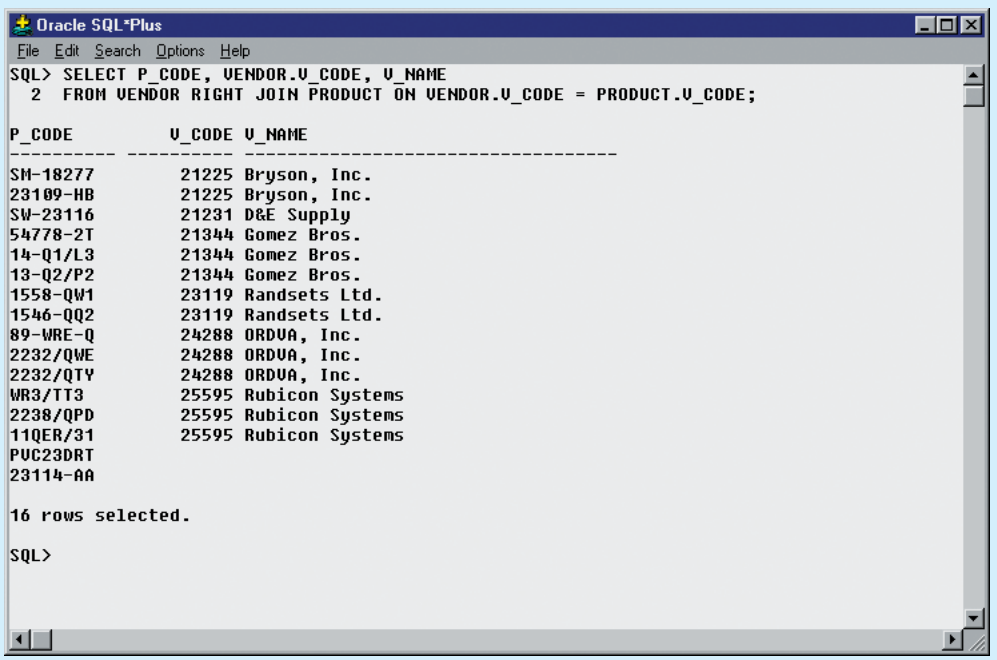
```
Oracle SQL*Plus
File  Edit  Search  Options  Help
SQL> SELECT P_CODE, VENDOR.V_CODE, V_NAME
  2  FROM VENDOR LEFT JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;

P_CODE         V_CODE V_NAME
---------- ---------- ------------------------------------
11QER/31        25595 Rubicon Systems
13-Q2/P2        21344 Gomez Bros.
14-Q1/L3        21344 Gomez Bros.
1546-QQ2        23119 Randsets Ltd.
1558-QW1        23119 Randsets Ltd.
2232/QTY        24288 ORDVA, Inc.
2232/QWE        24288 ORDVA, Inc.
2238/QPD        25595 Rubicon Systems
23109-HB        21225 Bryson, Inc.
54778-2T        21344 Gomez Bros.
89-WRE-Q        24288 ORDVA, Inc.
SM-18277        21225 Bryson, Inc.
SW-23116        21231 D&E Supply
WR3/TT3         25595 Rubicon Systems
                22567 Dome Supply
                21226 SuperLoo, Inc.
                24004 Brackman Bros.
                25501 Damal Supplies
                25443 B&K, Inc.

19 rows selected.

SQL>
```

**FIGURE 8.11** RIGHT JOIN results



```
Oracle SQL*Plus
File  Edit  Search  Options  Help
SQL> SELECT P_CODE, VENDOR.V_CODE, V_NAME
  2  FROM VENDOR RIGHT JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;

P_CODE         V_CODE V_NAME
---------- ---------- ------------------------------------
SM-18277        21225 Bryson, Inc.
23109-HB        21225 Bryson, Inc.
SW-23116        21231 D&E Supply
54778-2T        21344 Gomez Bros.
14-Q1/L3        21344 Gomez Bros.
13-Q2/P2        21344 Gomez Bros.
1558-QW1        23119 Randsets Ltd.
1546-QQ2        23119 Randsets Ltd.
89-WRE-Q        24288 ORDVA, Inc.
2232/QWE        24288 ORDVA, Inc.
2232/QTY        24288 ORDVA, Inc.
WR3/TT3         25595 Rubicon Systems
2238/QPD        25595 Rubicon Systems
11QER/31        25595 Rubicon Systems
PVC23DRT
23114-AA

16 rows selected.

SQL>
```

The full outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column) but also all of the rows with unmatched values in either side table. The syntax is:

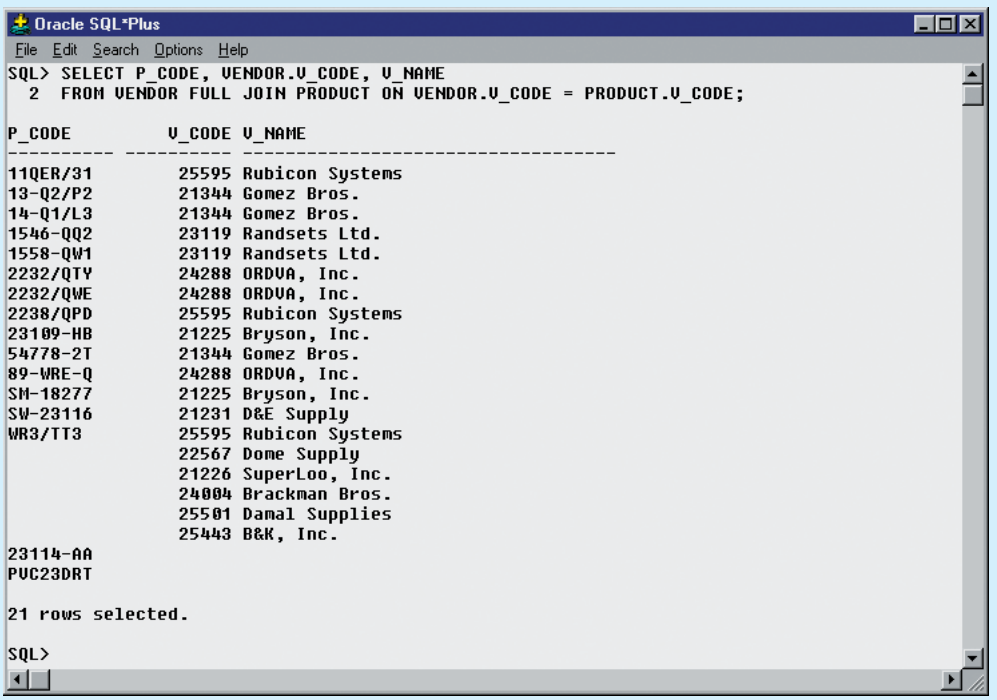
SELECT column-list
FROM *table1* FULL [OUTER] JOIN *table2* ON *join-condition*

For example, the following query lists the product code, vendor code, and vendor name for all products and includes all product rows (products without matching vendors) as well as all vendor rows (vendors without matching products):

```
SELECT    P_CODE, VENDOR.V_CODE, V_NAME
FROM      VENDOR FULL JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;
```

The SQL code and its results are shown in Figure 8.12.

**FIGURE 8.12 FULL JOIN results**




## 8.3 SUBQUERIES AND CORRELATED QUERIES

The use of joins in a relational database allows you to get information from two or more tables. For example, the following query allows you to get the customer's data with their respective invoices by joining the CUSTOMER and INVOICE tables.

```
SELECT    INV_NUMBER, INVOICE.CUS_CODE, CUS_LNAME, CUS_FNAME
FROM      CUSTOMER, INVOICE
WHERE     CUSTOMER.CUS_CODE = INVOICE.CUS_CODE;
```

In the previous query, the data from both tables (CUSTOMER and INVOICE) are processed at once, matching rows with shared CUS_CODE values.

However, it is often necessary to process data based on *other* processed data. Suppose, for example, that you want to generate a list of vendors who provide products. (Recall that not all vendors in the VENDOR table have provided products—some of them are only *potential* vendors.) In Chapter 7, you learned that you could generate such a list by writing the following query:

```
SELECT      V_CODE, V_NAME FROM VENDOR
WHERE       V_CODE NOT IN (SELECT V_CODE FROM PRODUCT);
```

Similarly, to generate a list of all products with a price greater than or equal to the average product price, you can write the following query:

```
SELECT      P_CODE, P_PRICE FROM PRODUCT
WHERE       P_PRICE >= (SELECT AVG(P_PRICE) FROM PRODUCT);
```

In both of those cases, you needed to get information that was not previously known:

- What vendors provide products?
- What is the average price of all products?

In both cases, you used a subquery to generate the required information that could then be used as input for the originating query.

You learned how to use subqueries in Chapter 7; let's review their basic characteristics:

- A subquery is a query (SELECT statement) inside a query.
- A subquery is normally expressed inside parentheses.
- The first query in the SQL statement is known as the outer query.
- The query inside the SQL statement is known as the inner query.
- The inner query is executed first.
- The output of an inner query is used as the input for the outer query.
- The entire SQL statement is sometimes referred to as a nested query.

In this section, you learn more about the practical use of subqueries. You already know that a subquery is based on the use of the SELECT statement to return one or more values to another query. But subqueries have a wide range of uses. For example, you can use a subquery within a SQL data manipulation language (DML) statement (such as INSERT, UPDATE, or DELETE) where a value or a list of values (such as multiple vendor codes or a table) is expected. Table 8.2 uses simple examples to summarize the use of SELECT subqueries in DML statements.

**TABLE 8.2 SELECT Subquery Examples**

| SELECT SUBQUERY EXAMPLES | EXPLANATION |
|---|---|
| INSERT INTO PRODUCT<br>SELECT * FROM P; | Inserts all rows from Table P into the PRODUCT table. Both tables must have the same attributes. The subquery returns all rows from Table P. |
| UPDATE PRODUCT<br>SET P_PRICE = (SELECT AVG(P_PRICE)<br>FROM PRODUCT)<br>WHERE V_CODE IN (SELECT V_CODE<br>FROM VENDOR<br>WHERE V_AREACODE = '615') | Updates the product price to the average product price, but only for the products that are provided by vendors who have an area code equal to 615. The first subquery returns the average price; the second subquery returns the list of vendors with an area code equal to 615. |
| DELETE FROM PRODUCT<br>WHERE V_CODE IN (SELECT V_CODE<br>FROM VENDOR<br>WHERE V_AREACODE = '615') | Deletes the PRODUCT table rows that are provided by vendors with area code equal to 615. The subquery returns the list of vendors codes with an area code equal to 615. |

Using the examples shown in Table 8.2, note that the subquery is always at the right side of a comparison or assigning expression. Also, a subquery can return one value or multiple values. To be precise, the subquery can return:

- *One single value (one column and one row).* This subquery is used anywhere a single value is expected, as in the right side of a comparison expression (such as in the preceding UPDATE example when you assign the average price to the product's price). Obviously, when you assign a value to an attribute, that value is a single value, not a list of values. Therefore, the subquery must return only one value (one column, one row). If the query returns multiple values, the DBMS will generate an error.
- *A list of values (one column and multiple rows).* This type of subquery is used anywhere a list of values is expected, such as when using the IN clause (that is, when comparing the vendor code to a list of vendors). Again, in this case, there is only one column of data with multiple value instances. This type of subquery is used frequently in combination with the IN operator in a WHERE conditional expression.
- *A virtual table (multicolumn, multirow set of values).* This type of subquery can be used anywhere a table is expected, such as when using the FROM clause. You will see this type of query later in this chapter.

It's important to note that a subquery can return no values at all; it is a NULL. In such cases, the output of the outer query might result in an error or a null empty set, depending where the subquery is used (in a comparison, an expression, or a table set).

In the following sections, you will learn how to write subqueries within the SELECT statement to retrieve data from the database.

### 8.3.1 WHERE Subqueries

The most common type of subquery uses an inner SELECT subquery on the right side of a WHERE comparison expression. For example, to find all products with a price greater than or equal to the average product price, you write the following query:

```
SELECT    P_CODE, P_PRICE FROM PRODUCT
WHERE     P_PRICE >= (SELECT AVG(P_PRICE) FROM PRODUCT);
```

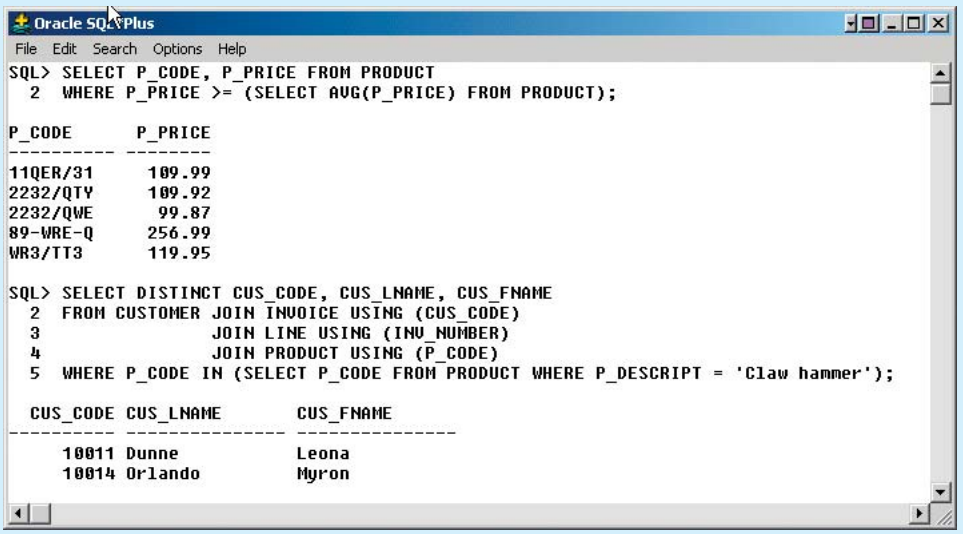
The output of the preceding query is shown in Figure 8.13. Note that this type of query, when used in a >, <, =, >=, or <= conditional expression, requires a subquery that returns only one single value (one column, one row). The value generated by the subquery must be of a "comparable" data type; if the attribute to the left of the comparison symbol is a character type, the subquery must return a character string. Also, if the query returns more than a single value, the DBMS will generate an error.

Subqueries can also be used in combination with joins. For example, the following query lists all of the customers who ordered the product "Claw hammer":

```
SELECT    DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
FROM      CUSTOMER JOIN INVOICE USING (CUS_CODE)
                   JOIN LINE USING (INV_NUMBER)
                   JOIN PRODUCT USING (P_CODE)
WHERE     P_CODE = (SELECT P_CODE FROM PRODUCT WHERE P_DESCRIPT = 'Claw hammer');
```

The result of that query is also shown in Figure 8.13.

**FIGURE 8.13** WHERE subquery example




In the preceding example, the inner query finds the P_CODE for the product "Claw hammer." The P_CODE is then used to restrict the selected rows to only those where the P_CODE in the LINE table matches the P_CODE for "Claw hammer." Note that the previous query could have been written this way:

```
SELECT    DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
FROM      CUSTOMER JOIN INVOICE USING (CUS_CODE)
                   JOIN LINE USING (INV_NUMBER)
                   JOIN PRODUCT USING (P_CODE)
WHERE     P_DESCRIPT = 'Claw hammer';
```

But what happens if the original query encounters the "Claw hammer" string in more than one product description? You get an error message. To compare one value to a list of values, you must use an IN operand, as shown in the next section.

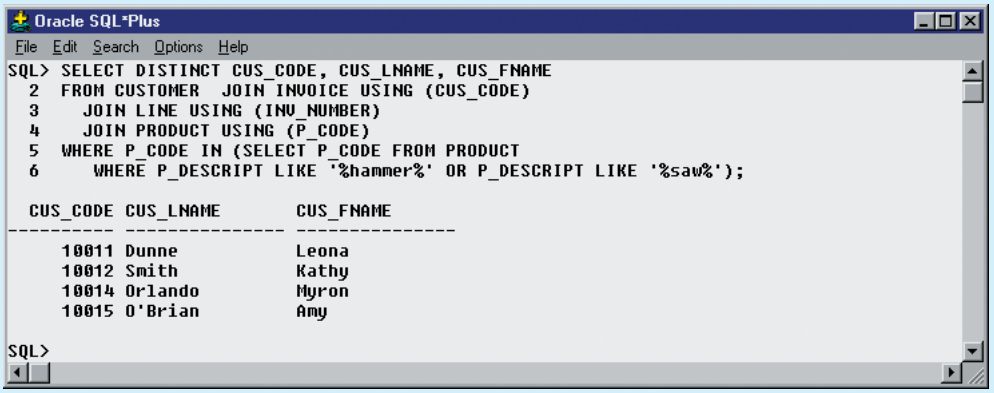
### 8.3.2 IN Subqueries

What would you do if you wanted to find all customers who purchased a "hammer" or any kind of saw or saw blade? Note that the product table has two different types of hammers: "Claw hammer" and "Sledge hammer." Also note that there are multiple occurrences of products that contain "saw" in their product descriptions. There are saw blades, jigsaws, and so on. In such cases, you need to compare the P_CODE not to one product code (single value) but to a list of product code values. When you want to compare a single attribute to a list of values, you use the IN operator. When the P_CODE values are not known beforehand, but they can be derived using a query, you must use an IN subquery. The following example lists all customers who have purchased hammers, saws, or saw blades.

```
SELECT    DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
FROM      CUSTOMER JOIN INVOICE USING (CUS_CODE)
                   JOIN LINE USING (INV_NUMBER)
                   JOIN PRODUCT USING (P_CODE)
WHERE     P_CODE IN ( SELECT     P_CODE FROM PRODUCT
                      WHERE      P_DESCRIPT LIKE '%hammer%'
                      OR         P_DESCRIPT LIKE '%saw%');
```

The result of that query is shown in Figure 8.14.

**FIGURE 8.14** IN subquery example



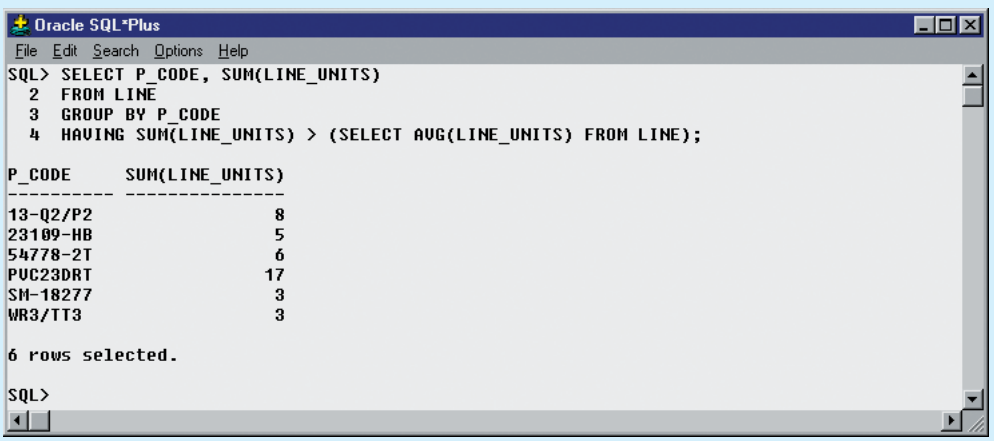

## 8.3.3 HAVING Subqueries

Just as you can use subqueries with the WHERE clause, you can use a subquery with a HAVING clause. Remember that the HAVING clause is used to restrict the output of a GROUP BY query by applying a conditional criteria to the grouped rows. For example, to list all products with the total quantity sold greater than the average quantity sold, you would write the following query:

```
SELECT    P_CODE, SUM(LINE_UNITS)
FROM      LINE
GROUP BY  P_CODE
HAVING    SUM(LINE_UNITS) > (SELECT AVG(LINE_UNITS) FROM LINE);
```

The result of that query is shown in Figure 8.15.

**FIGURE 8.15** HAVING subquery example



### 8.3.4 MULTIROW SUBQUERY OPERATORS: ANY AND ALL

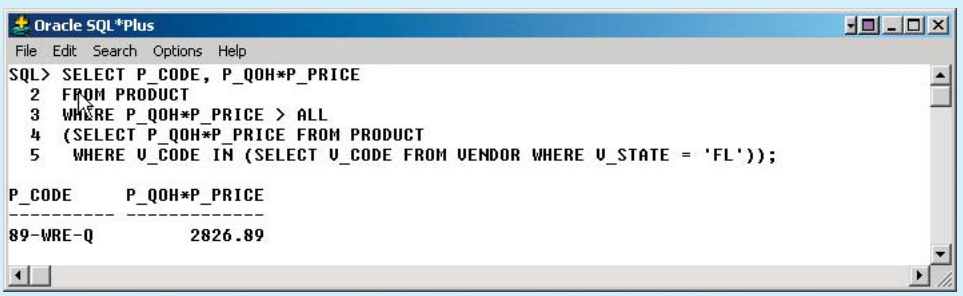
So far, you have learned that you must use an IN subquery when you need to compare a value to a list of values. But the IN subquery uses an equality operator; that is, it selects only those rows that match (are equal to) at least one of the values in the list. What happens if you need to do an inequality comparison (> or <) of one value to a list of values?

For example, suppose that you want to know which products have a product cost that is greater than all individual product costs for products provided by vendors from Florida.

```
SELECT    P_CODE, P_QOH * P_PRICE
FROM      PRODUCT
WHERE     P_QOH * P_PRICE > ALL (SELECT P_QOH * P_PRICE
                                 FROM PRODUCT
                                 WHERE V_CODE IN (SELECT V_CODE
                                                  FROM VENDOR
                                                  WHERE V_STATE = 'FL'));
```

The result of that query is shown in Figure 8.16.

**FIGURE 8.16** Multirow subquery operator example




It's important to note the following points about the query and its output in Figure 8.16:

- The query is a typical example of a nested query.
- The query has one outer SELECT statement with a SELECT subquery (call it $sq^A$) containing a second SELECT subquery (call it $sq^B$).
- The last SELECT subquery ($sq^B$) is executed first and returns a list of all vendors from Florida.
- The first SELECT subquery ($sq^A$) uses the output of the SELECT subquery ($sq^B$). The $sq^A$ subquery returns the list of product costs for all products provided by vendors from Florida.
- The use of the ALL operator allows you to compare a single value (P_QOH * P_PRICE) with a list of values returned by the first subquery ($sq^A$) using a comparison operator other than equals.
- For a row to appear in the result set, it has to meet the criterion P_QOH * P_PRICE > ALL, of the individual values returned by the subquery $sq^A$. The values returned by $sq^A$ are a list of product costs. In fact, "greater than ALL" is equivalent to "greater than the highest product cost of the list." In the same way, a condition of "less than ALL" is equivalent to "less than the lowest product cost of the list."

Another powerful operator is the ANY multirow operator (the near cousin of the ALL multirow operator). The ANY operator allows you to compare a single value to a list of values, selecting only the rows for which the inventory cost is greater than any value of the list or less than any value of the list. You could use the equal to ANY operator, which would be the equivalent of the IN operator.

### 8.3.5 FROM Subqueries

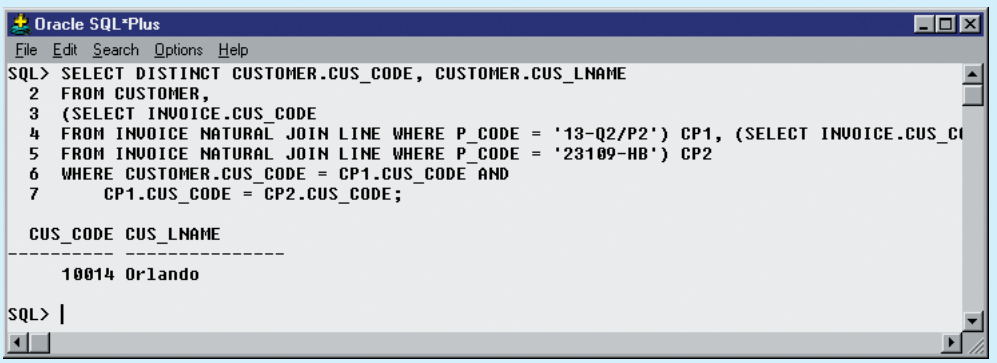
So far you have seen how the SELECT statement uses subqueries within WHERE, HAVING, and IN statements and how the ANY and ALL operators are used for multirow subqueries. In all of those cases, the subquery was part of a conditional expression and it always appeared at the right side of the expression. In this section, you will learn how to use subqueries in the FROM clause.

As you already know, the FROM clause specifies the table(s) from which the data will be drawn. Because the output of a SELECT statement is another table (or more precisely a "virtual" table), you could use a SELECT subquery in the FROM clause. For example, assume that you want to know all customers who have purchased products 13-Q2/P2 *and* 23109-HB. All product purchases are stored in the LINE table. It is easy to find out who purchased any given product by searching the P_CODE attribute in the LINE table. But in this case, you want to know all customers who purchased both products, not just one. You could write the following query:

```
SELECT    DISTINCT CUSTOMER.CUS_CODE, CUSTOMER.CUS_LNAME
FROM      CUSTOMER,
          (SELECT INVOICE.CUS_CODE FROM INVOICE NATURAL JOIN LINE
          WHERE P_CODE = '13-Q2/P2') CP1,
          (SELECT INVOICE.CUS_CODE FROM INVOICE NATURAL JOIN LINE
          WHERE P_CODE = '23109-HB') CP2
WHERE     CUSTOMER.CUS_CODE = CP1.CUS_CODE AND CP1.CUS_CODE = CP2.CUS_CODE;
```

The result of that query is shown in Figure 8.17.

**FIGURE 8.17 FROM subquery example**




Note in Figure 8.17 that the first subquery returns all customers who purchased product 13-Q2/P2, while the second subquery returns all customers who purchased product 23109-HB. So in this FROM subquery, you are joining the CUSTOMER table with two virtual tables. The join condition selects only the rows with matching CUS_CODE values in each table (base or virtual).

In the previous chapter, you learned that a view is also a virtual table; therefore, you can use a view name anywhere a table is expected. So in this example, you could create two views: one listing all customers who purchased product 13-Q2/P2 and another listing all customers who purchased product 23109-HB. Doing so, you would write the query as:

```
CREATE VIEW CP1 AS
        SELECT    INVOICE.CUS_CODE FROM INVOICE NATURAL JOIN LINE
        WHERE     P_CODE = '13-Q2/P2';
```

```
CREATE VIEW CP2 AS
        SELECT      INVOICE.CUS_CODE FROM INVOICE NATURAL JOIN LINE
        WHERE       P_CODE = '23109-HB';
SELECT      DISTINCT CUS_CODE, CUS_LNAME
FROM        CUSTOMER NATURAL JOIN CP1 NATURAL JOIN CP2;
```

You might speculate that the preceding query could also be written using the following syntax:

```
SELECT      CUS_CODE, CUS_LNAME
FROM        CUSTOMER NATURAL JOIN INVOICE NATURAL JOIN LINE
WHERE       P_CODE = '13-Q2/P2' AND P_CODE = '23109-HB';
```

But if you examine that query carefully, you will note that a P_CODE cannot be equal to two different values at the same time. Therefore, the query will not return any rows.

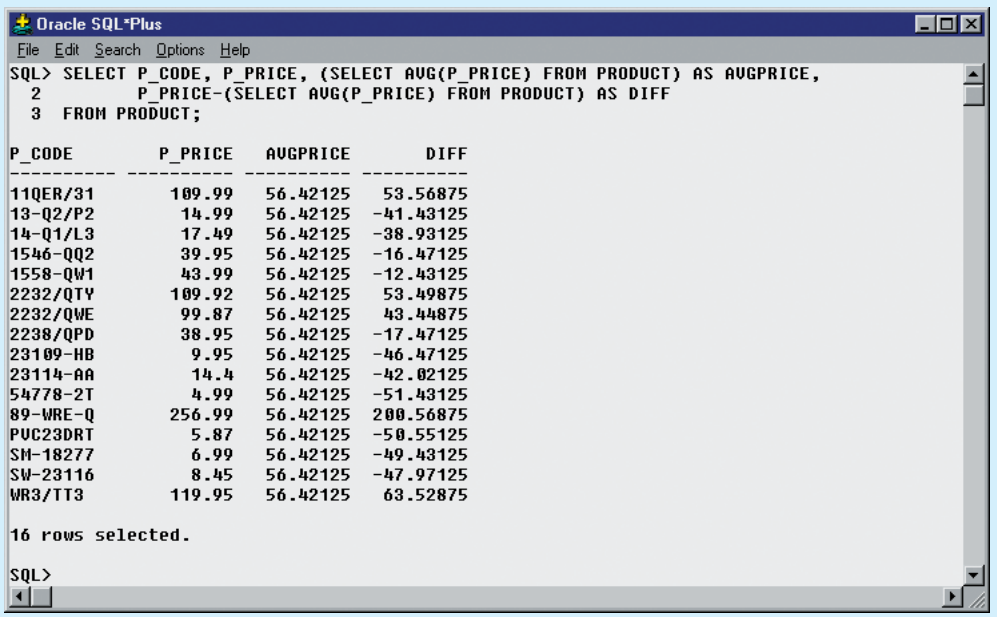
### 8.3.6 Attribute List Subqueries

The SELECT statement uses the attribute list to indicate what columns to project in the resulting set. Those columns can be attributes of base tables, computed attributes, or the result of an aggregate function. The attribute list can also include a subquery expression, also known as an inline subquery. A subquery in the attribute list must return one single value; otherwise, an error code is raised. For example, a simple inline query can be used to list the difference between each product's price and the average product price:

```
SELECT      P_CODE, P_PRICE, (SELECT AVG(P_PRICE) FROM PRODUCT) AS AVGPRICE,
            P_PRICE – (SELECT AVG(P_PRICE) FROM PRODUCT) AS DIFF
FROM        PRODUCT;
```

Figure 8.18 shows the result of that query.

**FIGURE 8.18** Inline subquery example



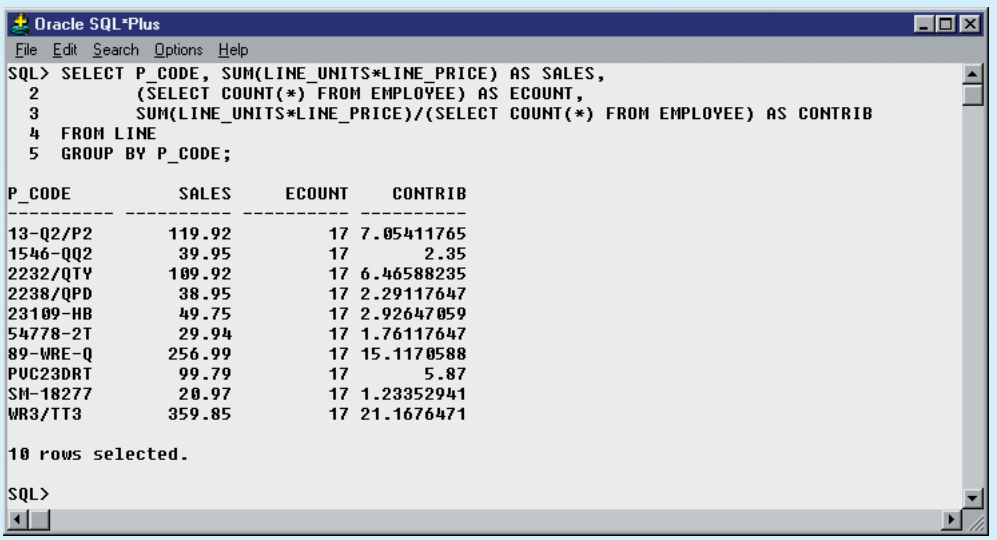
In Figure 8.18, note that the inline query output returns one single value (the average product's price) and that the value is the same in every row. Note also that the query used the full expression instead of the column aliases when computing the difference. In fact, if you try to use the alias in the difference expression, you will get an error message. The column alias cannot be used in computations in the attribute list when the alias is defined in the same attribute list. That DBMS requirement is the result of the way the DBMS parses and executes queries.

Another example will help you understand the use of attribute list subqueries and column aliases. For example, suppose that you want to know the product code, the total sales by product, and the contribution by employee of each product's sales. To get the sales by-product, you need to use only the LINE table. To compute the contribution by employee, you need to know the number of employees (from the EMPLOYEE table). As you study the tables' structures, you can see that the LINE and EMPLOYEE tables do not share a common attribute. In fact, you don't need a common attribute. You only need to know the total number of employees, not the total employees related to each product. So to answer the query, you would write the following code:

```
SELECT      P_CODE, SUM(LINE_UNITS * LINE_PRICE) AS SALES,
            (SELECT COUNT(*) FROM EMPLOYEE) AS ECOUNT,
            SUM(LINE_UNITS * LINE_PRICE)/(SELECT COUNT(*) FROM EMPLOYEE) AS CONTRIB
FROM        LINE
GROUP BY    P_CODE;
```

The result of that query is shown in Figure 8.19.

**FIGURE 8.19** **Another example of an inline subquery**




As you can see in Figure 8.19, the number of employees remains the same for each row in the result set. The use of that type of subquery is limited to certain instances where you need to include data from other tables that are not directly related to a main table or tables in the query. The value will remain the same for each row, like a constant in a programming language. (You will learn another use of inline subqueries in Section 8.3.7, Correlated Subqueries.) Note that you cannot use an alias in the attribute list to write the expression that computes the contribution per employee.

Another way to write the same query by using column aliases requires the use of a subquery in the FROM clause, as follows:

```
SELECT    P_CODE, SALES, ECOUNT, SALES/ECOUNT AS CONTRIB
FROM      (SELECT P_CODE, SUM(LINE_UNITS * LINE_PRICE) AS SALES,
                  (SELECT COUNT(*) FROM EMPLOYEE) AS ECOUNT
          FROM    LINE
          GROUP BY P_CODE);
```

In that case, you are actually using two subqueries. The subquery in the FROM clause executes first and returns a virtual table with three columns: P_CODE, SALES, and ECOUNT. The FROM subquery contains an inline subquery that returns the number of employees as ECOUNT. Because the outer query receives the output of the inner query, you can now refer to the columns in the outer subquery by using the column aliases.

### 8.3.7 Correlated Subqueries

Until now, all subqueries you have learned execute independently. That is, each subquery in a command sequence executes in a serial fashion, one after another. The inner subquery executes first; its output is used by the outer query, which then executes until the last outer query executes (the first SQL statement in the code).

In contrast, a **correlated subquery** is a subquery that executes once for each row in the outer query. That process is similar to the typical nested loop in a programming language. For example:

```
FOR X = 1 TO 2
        FOR Y = 1 TO 3
                PRINT "X = "X, "Y = "Y
        END
END
```

will yield the output:

```
X = 1       Y = 1
X = 1       Y = 2
X = 1       Y = 3
X = 2       Y = 1
X = 2       Y = 2
X = 2       Y = 3
```

Note that the outer loop X = 1 TO 2 begins the process by setting X = 1 and then the inner loop Y = 1 TO 3 is completed for each X outer loop value. The relational DBMS uses the same sequence to produce correlated subquery results:

1. It initiates the outer query.
2. For each row of the outer query result set, it executes the inner query by passing the outer row to the inner query.

That process is the opposite of that of the subqueries as you have already seen. The query is called a *correlated* subquery because the inner query is *related* to the outer query by the fact that the inner query references a column of the outer subquery.

To see the correlated subquery in action, suppose that you want to know all product sales in which the units sold value is greater than the average units sold value *for that product* (as opposed to the average for *all* products). In that case, the following procedure must be completed:

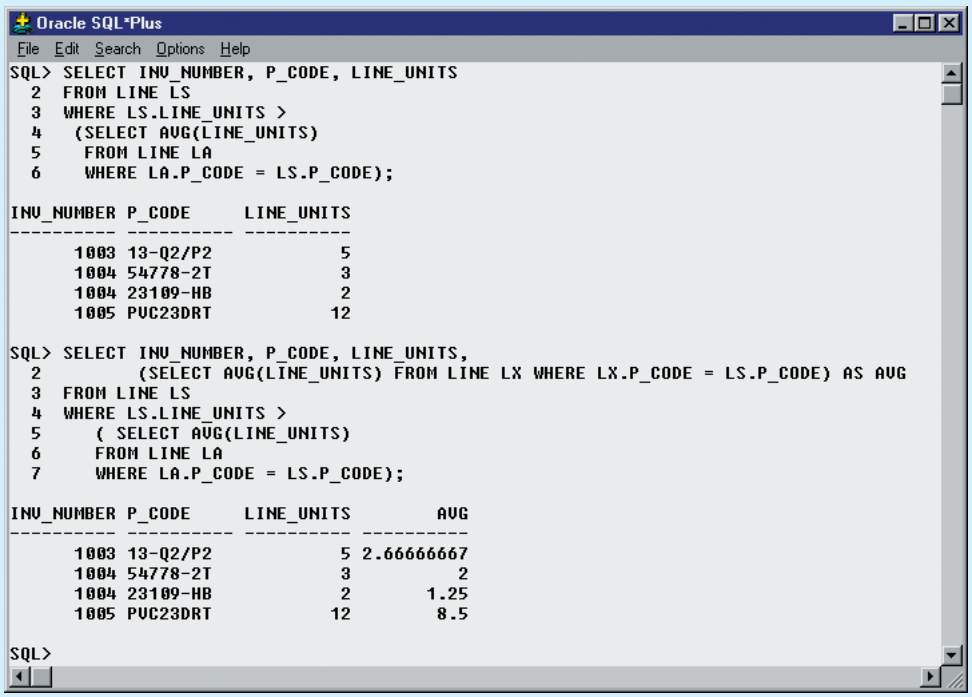
1. Compute the average-units-sold value for a product.
2. Compare the average computed in Step 1 to the units sold in each sale row and then select only the rows in which the number of units sold is greater.

The following correlated query completes the preceding two-step process:

```
SELECT      INV_NUMBER, P_CODE, LINE_UNITS
FROM        LINE LS
WHERE       LS.LINE_UNITS > (SELECT     AVG(LINE_UNITS)
                            FROM       LINE LA
                            WHERE      LA.P_CODE = LS.P_CODE);
```

The first example in Figure 8.20 shows the result of that query.

**FIGURE 8.20** **Correlated subquery examples**



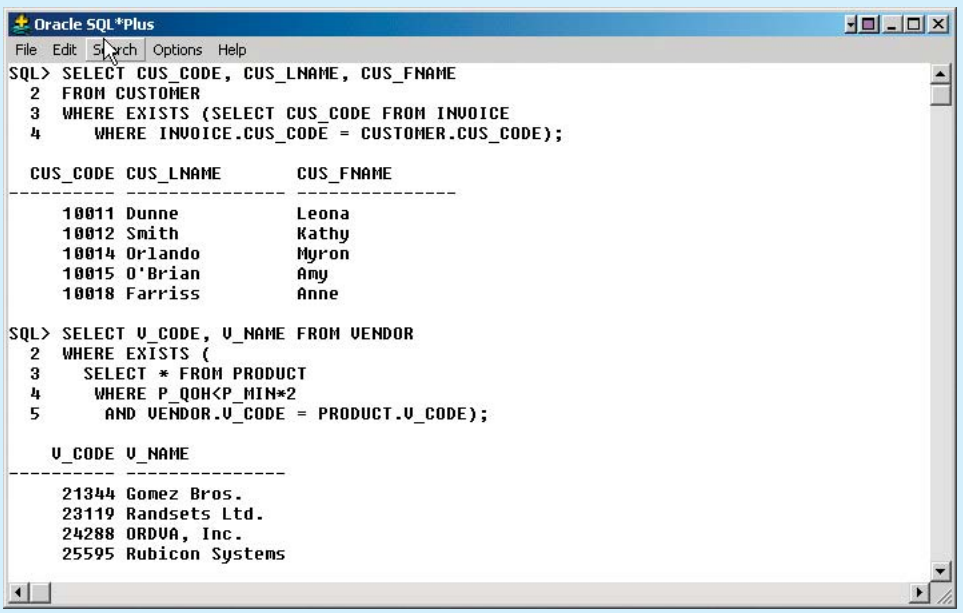

In the top query and its result in Figure 8.20, note that the LINE table is used more than once, so you must use table aliases. In that case, the inner query computes the average units sold of the product that matches the P_CODE of the outer query P_CODE. That is, the inner query runs once, using the first product code found in the (outer) LINE table, and returns the average sale for that product. When the number of units sold in that (outer) LINE row is greater than the average computed, the row is added to the output. Then the inner query runs again, this time using the second product code found in the (outer) LINE table. The process repeats until the inner query has run for all rows in the (outer) LINE table. In that case, the inner query will be repeated as many times as there are rows in the outer query.

To verify the results and to provide an example of how you can combine subqueries, you can add a correlated inline subquery to the previous query. That correlated inline subquery will show the average units sold column for each product. (See the second query and its results in Figure 8.20.) As you can see, the new query contains a correlated inline subquery that computes the average units sold for each product. You not only get an answer, but you can also verify that the answer is correct.

Correlated subqueries can also be used with the EXISTS special operator. For example, suppose that you want to know all customers who have placed an order lately. In that case, you could use a correlated subquery like the first one shown in Figure 8.21:

```
SELECT    CUS_CODE, CUS_LNAME, CUS_FNAME
FROM      CUSTOMER
WHERE     EXISTS    (SELECT    CUS_CODE FROM INVOICE
                    WHERE     INVOICE.CUS_CODE = CUSTOMER.CUS_CODE);
```

**FIGURE 8.21** **EXISTS correlated subquery examples**




The second example of an EXISTS correlated subquery in Figure 8.21 will help you understand how to use correlated queries. For example, suppose that you want to know what vendors you must contact to start ordering products that are approaching the minimum quantity-on-hand value. In particular, you want to know the vendor code and name of vendors for products having a quantity on hand that is less than double the minimum quantity. The query that answers that question is as follows:

```
SELECT    V_CODE, V_NAME
FROM      VENDOR
WHERE     EXISTS    (SELECT    *
                    FROM      PRODUCT
                    WHERE     P_QOH < P_MIN * 2
                    AND       VENDOR.V_CODE = PRODUCT.V_CODE);
```

In the second query in Figure 8.21, note that:

1. The inner correlated subquery runs using the first vendor.
2. If any products match the condition (quantity on hand is less than double the minimum quantity), the vendor code and name are listed in the output.
3. The correlated subquery runs using the second vendor, and the process repeats itself until all vendors are used.

## 8.4 SQL FUNCTIONS

The data in databases are the basis of critical business information. Generating information from data often requires many data manipulations. Sometimes such data manipulation involves the decomposition of data elements. For example, an employee's date of birth can be subdivided into a day, a month, and a year. A product manufacturing code (for example, SE-05-2-09-1234-1-3/12/04-19:26:48) can be designed to record the manufacturing region, plant, shift, production line, employee number, date, and time. For years, conventional programming languages have had special functions that enabled programmers to perform data transformations like those data decompositions. If you know a modern programming language, it's very likely that the SQL functions in this section will look familiar.

SQL functions are very useful tools. You'll need to use functions when you want to list all employees ordered by year of birth or when your marketing department wants you to generate a list of all customers ordered by zip code and the first three digits of their telephone numbers. In both of those cases, you'll need to use data elements that are not present as such in the database; instead, you'll need a SQL function that can be derived from an existing attribute. Functions always use a numerical, date, or string value. The value may be part of the command itself (a constant or literal) or it may be an attribute located in a table. Therefore, a function may appear anywhere in an SQL statement where a value or an attribute can be used.

There are many types of SQL functions, such as arithmetic, trigonometric, string, date, and time functions. This section will not explain all of those types of functions in detail, but it will give you a brief overview of the most useful ones.

**NOTE**

Although the main DBMS vendors support the SQL functions covered here, the syntax or degree of support will probably differ. In fact, DBMS vendors invariably add their own functions to products to lure new customers. The functions covered in this section represent just a small portion of functions supported by your DBMS. Read your DBMS SQL reference manual for a complete list of available functions.

### 8.4.1 DATE AND TIME FUNCTIONS

All SQL-standard DBMSs support date and time functions. All date functions take one parameter (of a date or character data type) and return a value (character, numeric, or date type). Unfortunately, date/time data types are implemented differently by different DBMS vendors. The problem occurs because the ANSI SQL standard defines date data types, but it does not say how those data types are to be stored. Instead, it lets the vendor deal with that issue.

Because date/time functions differ from vendor to vendor, this section will cover basic date/time functions for MS Access/SQL Server and for Oracle. Table 8.3 shows a list of selected MS Access/SQL Server date/time functions.

**TABLE 8.3** **Selected MS Access/SQL Server Date/Time Functions**

| FUNCTION | EXAMPLE(S) |
|---|---|
| **YEAR**<br>Returns a four-digit year<br>Syntax:<br>YEAR(date_value) | Lists all employees born in 1966:<br>SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,<br>YEAR(EMP_DOB) AS YEAR<br>FROM EMPLOYEE<br>WHERE YEAR(EMP_DOB) = 1966; |
| **MONTH**<br>Returns a two-digit month code<br>Syntax:<br>MONTH(date_value) | Lists all employees born in November:<br>SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,<br>MONTH(EMP_DOB) AS MONTH<br>FROM EMPLOYEE<br>WHERE MONTH(EMP_DOB) = 11; |
| **DAY**<br>Returns the number of the day<br>Syntax:<br>DAY(date_value) | Lists all employees born on the 14th day of the month:<br>SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,<br>DAY(EMP_DOB) AS DAY<br>FROM EMPLOYEE<br>WHERE DAY(EMP_DOB) = 14; |
| **DATE() – MS Access**<br>**GETDATE() – SQL Server**<br>Returns today's date | Lists how many days are left until Christmas:<br>SELECT #25-Dec-2010# – DATE();<br>Note two features:<br>• There is no FROM clause, which is acceptable in MS Access.<br>• The Christmas date is enclosed in # signs because you are doing date arithmetic.<br>In MS SQL Server:<br>Use GETDATE() to get the current system date. To compute the difference between dates, use the DATEDIFF function (see below). |
| **DATEADD – SQL Server**<br>Adds a number of selected time periods to a date<br>Syntax:<br>**DATEADD(datepart, number, date)** | Adds a *number* of *dateparts* to a given *date*. Dateparts can be minutes, hours, days, weeks, months, quarters, or years. For example:<br>SELECT DATEADD(day,90, P_INDATE) AS DueDate<br>FROM PRODUCT;<br>The preceding example adds 90 days to P_INDATE.<br>In MS Access use:<br>SELECT P_INDATE+90 AS DueDate<br>FROM PRODUCT; |
| **DATEDIFF – SQL Server**<br>Subtracts two dates<br>Syntax:<br>**DATEDIFF(datepart, startdate, enddate)** | Returns the difference between two dates expressed in a selected *datepart*. For example:<br>SELECT DATEDIFF(day, P_INDATE, GETDATE()) AS DaysAgo<br>FROM PRODUCT;<br>In MS Access use:<br>SELECT DATE() - P_INDATE AS DaysAgo<br>FROM PRODUCT; |

Table 8.4 shows the equivalent date/time functions used in Oracle. Note that Oracle uses the same function (TO_CHAR) to extract the various parts of a date. Also, another function (TO_DATE) is used to convert character strings to a valid Oracle date format that can be used in date arithmetic.

**TABLE 8.4** Selected Oracle Date/Time Functions

| FUNCTION | EXAMPLE(S) |
|---|---|
| **TO_CHAR**<br>Returns a character string or a formatted string from a date value<br>Syntax:<br>TO_CHAR(date_value, fmt)<br>fmt = format used; can be:<br>MONTH: name of month<br>MON: three-letter month name<br>MM: two-digit month name<br>D: number for day of week<br>DD: number day of month<br>DAY: name of day of week<br>YYYY: four-digit year value<br>YY: two-digit year value | Lists all employees born in 1982:<br>SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,<br>TO_CHAR(EMP_DOB, 'YYYY') AS YEAR<br>FROM EMPLOYEE<br>WHERE TO_CHAR(EMP_DOB, 'YYYY') = '1982';<br>Lists all employees born in November:<br>SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,<br>TO_CHAR(EMP_DOB, 'MM') AS MONTH<br>FROM EMPLOYEE<br>WHERE TO_CHAR(EMP_DOB, 'MM') = '11';<br>Lists all employees born on the 14th day of the month:<br>SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,<br>TO_CHAR(EMP_DOB, 'DD') AS DAY<br>FROM EMPLOYEE<br>WHERE TO_CHAR(EMP_DOB, 'DD') = '14'; |
| **TO_DATE**<br>Returns a date value using a character string and a date format mask; also used to translate a date between formats<br>Syntax:<br>TO_DATE(char_value, fmt)<br>fmt = format used; can be:<br>MONTH: name of month<br>MON: three-letter month name<br>MM: two-digit month name<br>D: number for day of week<br>DD: number day of month<br>DAY: name of day of week<br>YYYY: four-digit year value<br>YY: two-digit year value | Lists the approximate age of the employees on the company's tenth anniversary date (11/25/2010):<br>SELECT EMP_LNAME, EMP_FNAME,<br>EMP_DOB, '11/25/2010' AS ANIV_DATE,<br>(TO_DATE('11/25/2000','MM/DD/YYYY') - EMP_DOB)/365 AS YEARS<br>FROM EMPLOYEE<br>ORDER BY YEARS;<br>Note the following:<br>• '11/25/2010' is a text string, not a date.<br>• The TO_DATE function translates the text string to a valid Oracle date used in date arithmetic.<br>How many days between Thanksgiving and Christmas 2008?<br>SELECT TO_DATE('2010/12/25','YYYY/MM/DD') –<br>TO_DATE('NOVEMBER 27, 2010','MONTH DD, YYYY')<br>FROM DUAL;<br>Note the following:<br>• The TO_DATE function translates the text string to a valid Oracle date used in date arithmetic.<br>• DUAL is Oracle's pseudo-table used only for cases where a table is not really needed. |
| **SYSDATE**<br>Returns today's date | Lists how many days are left until Christmas:<br>SELECT TO_DATE('25-Dec-2010','DD-MON-YYYY') SYSDATE<br>FROM DUAL;<br>Notice two things:<br>• DUAL is Oracle's pseudo-table used only for cases where a table is not really needed.<br>• The Christmas date is enclosed in a TO_DATE function to translate the date to a valid date format. |
| **ADD_MONTHS**<br>Adds a number of months to a date; useful for adding months or years to a date<br>Syntax:<br>ADD_MONTHS(date_value, n)<br>n = number of months | Lists all products with their expiration date (two years from the purchase date):<br>SELECT P_CODE, P_INDATE, ADD_MONTHS(P_INDATE,24)<br>FROM PRODUCT<br>ORDER BY ADD_MONTHS(P_INDATE,24); |

**TABLE 8.4** Selected Oracle Date/Time Functions (continued)

| FUNCTION | EXAMPLE(S) |
|---|---|
| **LAST_DAY**<br>Returns the date of the last day of the month given in a date<br>Syntax:<br>LAST_DAY(date_value) | Lists all employees who were hired within the last seven days of a month:<br>SELECT EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE<br>FROM EMPLOYEE<br>WHERE EMP_HIRE_DATE >=LAST_DAY(EMP_HIRE_DATE)-7; |

### 8.4.2 Numeric Functions

Numeric functions can be grouped in many different ways, such as algebraic, trigonometric, and logarithmic. In this section, you will learn two very useful functions. Do not confuse the SQL aggregate functions you saw in the previous chapter with the numeric functions in this section. The first group operates over a set of values (multiple rows—hence, the name *aggregate functions*), while the numeric functions covered here operate over a single row. Numeric functions take one numeric parameter and return one value. Table 8.5 shows a selected group of numeric functions available.

**TABLE 8.5** Selected Numeric Functions

| FUNCTION | EXAMPLE(S) |
|---|---|
| **ABS**<br>Returns the absolute value of a number<br>Syntax:<br>ABS(numeric_value) | In Oracle use:<br>SELECT 1.95, -1.93, ABS(1.95), ABS(-1.93)<br>FROM DUAL;<br>In MS Access/SQL Server use:<br>SELECT 1.95, -1.93, ABS(1.95), ABS(-1.93); |
| **ROUND**<br>Rounds a value to a specified precision (number of digits)<br>Syntax:<br>ROUND(numeric_value, p)<br>p = precision | Lists the product prices rounded to one and zero decimal places:<br>SELECT P_CODE, P_PRICE,<br>ROUND(P_PRICE,1) AS PRICE1,<br>ROUND(P_PRICE,0) AS PRICE0<br>FROM PRODUCT; |
| **CEIL/CEILING/FLOOR**<br>Returns the smallest integer greater than or equal to a number or returns the largest integer equal to or less than a number, respectively<br>Syntax:<br>CEIL(numeric_value) – Oracle<br>CEILING(numeric_value) – SQL Server<br>FLOOR(numeric_value) | Lists the product price, smallest integer greater than or equal to the product price, and the largest integer equal to or less than the product price.<br>In Oracle use:<br>SELECT P_PRICE, CEIL(P_PRICE), FLOOR(P_PRICE)<br>FROM PRODUCT;<br>In SQL Server use:<br>SELECT P_PRICE, CEILING(P_PRICE), FLOOR(P_PRICE)<br>FROM PRODUCT;<br>MS Access does not support these functions. |

### 8.4.3 String Functions

String manipulations are among the most-used functions in programming. If you have ever created a report using any programming language, you know the importance of properly concatenating strings of characters, printing names in uppercase, or knowing the length of a given attribute. Table 8.6 shows a subset of useful string manipulation functions.

**TABLE 8.6 Selected String Functions**

| FUNCTION | EXAMPLE(S) |
|---|---|
| **Concatenation**<br>**\|\| – Oracle**<br>**+ – MS Access/SQL Server**<br>Concatenates data from two different character columns and returns a single column<br>Syntax:<br>strg_value \|\| strg_value<br>strg_value + strg_value | Lists all employee names (concatenated).<br>In Oracle use:<br>SELECT EMP_LNAME \|\| ', ' \|\| EMP_FNAME AS NAME<br>FROM EMPLOYEE;<br>In MS Access / SQL Server use:<br>SELECT EMP_LNAME + ', ' + EMP_FNAME AS NAME<br>FROM EMPLOYEE; |
| **UPPER/LOWER**<br>Returns a string in all capital or all lowercase letters<br>Syntax:<br>UPPER(strg_value)<br>LOWER(strg_value) | Lists all employee names in all capital letters (concatenated).<br>In Oracle use:<br>SELECT UPPER(EMP_LNAME) \|\| ', ' \|\| UPPER(EMP_FNAME) AS NAME<br>FROM EMPLOYEE;<br>In SQL Server use:<br>SELECT UPPER(EMP_LNAME) + ', ' + UPPER(EMP_FNAME) AS NAME<br>FROM EMPLOYEE;<br>Lists all employee names in all lowercase letters (concatenated).<br>In Oracle use:<br>SELECT LOWER(EMP_LNAME) \|\| ', ' \|\| LOWER(EMP_FNAME) AS NAME<br>FROM EMPLOYEE;<br>In SQL Server use:<br>SELECT LOWER(EMP_LNAME) + ', ' + LOWER(EMP_FNAME) AS NAME<br>FROM EMPLOYEE;<br>Not supported by MS Access. |
| **SUBSTRING**<br>Returns a substring or part of a given string parameter<br>Syntax:<br>SUBSTR(strg_value, p, l) – Oracle<br>SUBSTRING(strg_value,p,l) – SQL Server<br>p = start position<br>l = length of characters | Lists the first three characters of all employee phone numbers.<br>In Oracle use:<br>SELECT EMP_PHONE, SUBSTR(EMP_PHONE,1,3) AS PREFIX<br>FROM EMPLOYEE;<br>In SQL Server use:<br>SELECT EMP_PHONE, SUBSTRING(EMP_PHONE,1,3) AS PREFIX<br>FROM EMPLOYEE;<br>Not supported by MS Access. |
| **LENGTH**<br>Returns the number of characters in a string value<br>Syntax:<br>LENGTH(strg_value) – Oracle<br>LEN(strg_value) – SQL Server | Lists all employee last names and the length of their names; ordered descended by last name length.<br>In Oracle use:<br>SELECT EMP_LNAME, LENGTH(EMP_LNAME) AS NAMESIZE<br>FROM EMPLOYEE;<br>In MS Access / SQL Server use:<br>SELECT EMP_LNAME, LEN(EMP_LNAME) AS NAMESIZE<br>FROM EMPLOYEE; |

### 8.4.4 Conversion Functions

Conversion functions allow you to take a value of a given data type and convert it to the equivalent value in another data type. In Section 8.4.1, you learned about two of the basic Oracle SQL conversion functions: TO_CHAR and TO_DATE. Note that the TO_CHAR function takes a date value and returns a character string representing a day, a month, or a year. In the same way, the TO_DATE function takes a character string representing a date and returns an actual date in Oracle format. SQL Server uses the CAST and CONVERT functions to convert one data type to another. A summary of the selected functions is shown in Table 8.7.

**TABLE 8.7 Selected Conversion Functions**

| FUNCTION | EXAMPLE(S) |
|---|---|
| **Numeric to Character:**<br>**TO_CHAR – Oracle**<br>**CAST – SQL Server**<br>**CONVERT – SQL Server**<br>Returns a character string from a numeric value.<br>Syntax:<br>Oracle: TO_CHAR(numeric_value, fmt)<br>SQL Server:<br>CAST (numeric AS varchar(length))<br>CONVERT(varchar(length), numeric) | Lists all product prices, quantity on hand, percent discount, and total inventory cost using formatted values.<br>In Oracle use:<br>SELECT P_CODE,<br>TO_CHAR(P_PRICE,'999.99') AS PRICE,<br>TO_CHAR(P_QOH,'9,999.99') AS QUANTITY,<br>TO_CHAR(P_DISCOUNT,'0.99') AS DISC,<br>TO_CHAR(P_PRICE*P_QOH,'99,999.99')<br>AS TOTAL_COST<br>FROM PRODUCT;<br>In SQL Server use:<br>SELECT P_CODE, CAST(P_PRICE AS VARCHAR(8)) AS PRICE,<br>CONVERT(VARCHAR(4),P_QOH) AS QUANTITY,<br>CAST(P_DISCOUNT AS VARCHAR(4)) AS DISC,<br>CAST(P_PRICE*P_QOH AS VARCHAR(10)) AS TOTAL_COST<br>FROM PRODUCT;<br>Not supported in MS Access. |
| **Date to Character:**<br>**TO_CHAR – Oracle**<br>**CAST – SQL Server**<br>**CONVERT – SQL Server**<br>Returns a character string or a formatted character string from a date value<br>Syntax:<br>Oracle: TO_CHAR(date_value, fmt)<br>SQL Server:<br>CAST (date AS varchar(length))<br>CONVERT(varchar(length), date) | Lists all employee dates of birth, using different date formats.<br>In Oracle use:<br>SELECT EMP_LNAME, EMP_DOB,<br>TO_CHAR(EMP_DOB, 'DAY, MONTH DD, YYYY')<br>AS 'DATEOFBIRTH'<br>FROM EMPLOYEE;<br>SELECT EMP_LNAME, EMP_DOB,<br>TO_CHAR(EMP_DOB, 'YYYY/MM/DD')<br>AS 'DATEOFBIRTH'<br>FROM EMPLOYEE;<br>In SQL Server use:<br>SELECT EMP_LNAME, EMP_DOB,<br>CONVERT(varchar(11),EMP_DOB) AS "DATE OF BIRTH"<br>FROM EMPLOYEE;<br>SELECT EMP_LNAME, EMP_DOB,<br>CAST(EMP_DOB as varchar(11)) AS "DATE OF BIRTH"<br>FROM EMPLOYEE;<br>Not supported in MS Access. |
| **String to Number:**<br>**TO_NUMBER**<br>Returns a formatted number from a character string, using a given format<br>Syntax:<br>Oracle:<br>TO_NUMBER(char_value, fmt)<br>fmt = format used; can be:<br>9 = displays a digit<br>0 = displays a leading zero<br>, = displays the comma<br>. = displays the decimal point<br>$ = displays the dollar sign<br>B = leading blank<br>S = leading sign<br>MI = trailing minus sign | Converts text strings to numeric values when importing data to a table from another source in text format; for example, the query shown below uses the TO_NUMBER function to convert text formatted to Oracle default numeric values using the format masks given.<br>In Oracle use:<br>SELECT TO_NUMBER('-123.99', 'S999.99'),<br>TO_NUMBER('99.78-','B999.99MI')<br>FROM DUAL;<br>In SQL Server use:<br>SELECT CAST('-123.99' AS NUMERIC(8,2)),<br>CAST('-99.78' AS NUMERIC(8,2))<br>The SQL Server CAST function does not support the trailing sign on the character string.<br>Not supported in MS Access. |

**TABLE 8.7** Selected Conversion Functions (continued)

| FUNCTION | EXAMPLE(S) |
|---|---|
| **CASE – SQL Server**<br>**DECODE – Oracle**<br>Compares an attribute or expression with a series of values and returns an associated value or a default value if no match is found<br>Syntax:<br>Oracle:<br>DECODE(*e, x, y, d*)<br>*e* = attribute or expression<br>*x* = value with which to compare *e*<br>*y* = value to return in *e* = *x*<br>*d* = default value to return if *e* is not equal to *x*<br>SQL Server:<br>CASE When *condition*<br>THEN value1 ELSE value2 END | The following example returns the sales tax rate for specified states:<br>• Compares V_STATE to 'CA'; if the values match, it returns .08.<br>• Compares V_STATE to 'FL'; if the values match, it returns .05.<br>• Compares V_STATE to 'TN'; if the values match, it returns .085.<br>If there is no match, it returns 0.00 (the default value).<br>SELECT V_CODE, V_STATE,<br>DECODE(V_STATE,'CA',.08,'FL',.05, 'TN',.085, 0.00)<br>AS TAX<br>FROM VENDOR;<br>In SQL Server use:<br>SELECT V_CODE, V_STATE,<br>CASE WHEN V_STATE = 'CA' THEN .08<br>WHEN V_STATE = 'FL' THEN .05<br>WHEN V_STATE = 'TN' THEN .085<br>ELSE 0.00 END AS TAX<br>FROM VENDOR<br>Not supported in MS Access. |

## 8.5 ORACLE SEQUENCES

If you use MS Access, you might be familiar with the AutoNumber data type, which you can use to define a column in your table that will be automatically populated with unique numeric values. In fact, if you create a table in MS Access and forget to define a primary key, MS Access will offer to create a primary key column; if you accept, you will notice that MS Access creates a column named *ID* with an AutoNumber data type. After you define a column as an AutoNumber type, every time you insert a row in the table, MS Access will automatically add a value to that column, starting with 1 and increasing the value by 1 in every new row you add. Also, you cannot include that column in your INSERT statements—Access will not let you edit that value at all. MS SQL Server uses the Identity column property to serve a similar purpose. In MS SQL Server a table can have at most one column defined as an Identity column. This column behaves similarly to an MS Access column with the AutoNumber data type.

Oracle does not support the AutoNumber data type or the Identity column property. Instead, you can use a "sequence" to assign values to a column on a table. But an Oracle sequence is very different from the Access AutoNumber data type and deserves close scrutiny:

- Oracle sequences are an independent object in the database. (Sequences are not a data type.)
- Oracle sequences have a name and can be used anywhere a value is expected.
- Oracle sequences are not tied to a table or a column.
- Oracle sequences generate a numeric value that can be assigned to any column in any table.
- The table attribute to which you assigned a value based on a sequence can be edited and modified.
- An Oracle sequence can be created and deleted anytime.

The basic syntax to create a sequence in Oracle is:

CREATE SEQUENCE *name* [START WITH *n*] [INCREMENT BY *n*] [CACHE | NOCACHE]

where:

- *name* is the name of the sequence.
- *n* is an integer value that can be positive or negative.
- *START WITH* specifies the initial sequence value. (The default value is 1.)

- *INCREMENT BY* determines the value by which the sequence is incremented. (The default increment value is 1. The sequence increment can be positive or negative to enable you to create ascending or descending sequences.)
- The *CACHE* or *NOCACHE* clause indicates whether Oracle will preallocate sequence numbers in memory. (Oracle preallocates 20 values by default.)

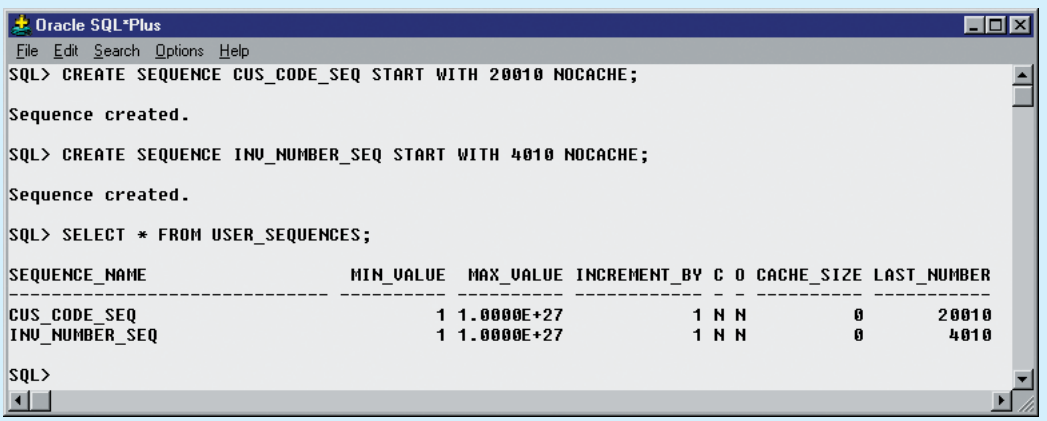
For example, you could create a sequence to automatically assign values to the customer code each time a new customer is added and create another sequence to automatically assign values to the invoice number each time a new invoice is added. The SQL code to accomplish those tasks is:

```
CREATE SEQUENCE CUS_CODE_SEQ START WITH 20010 NOCACHE;
CREATE SEQUENCE INV_NUMBER_SEQ START WITH 4010 NOCACHE;
```

You can check all of the sequences you have created by using the following SQL command, illustrated in Figure 8.22:

```
SELECT * FROM USER_SEQUENCES;
```

**FIGURE 8.22** **Oracle sequence**




To use sequences during data entry, you must use two special pseudo-columns: NEXTVAL and CURRVAL. NEXTVAL retrieves the next available value from a sequence, and CURRVAL retrieves the current value of a sequence. For example, you can use the following code to enter a new customer:

```
INSERT INTO CUSTOMER
VALUES (CUS_CODE_SEQ.NEXTVAL, 'Connery', 'Sean', NULL, '615', '898-2008', 0.00);
```

The preceding SQL statement adds a new customer to the CUSTOMER table and assigns the value 20010 to the CUS_CODE attribute. Let's examine some important sequence characteristics:

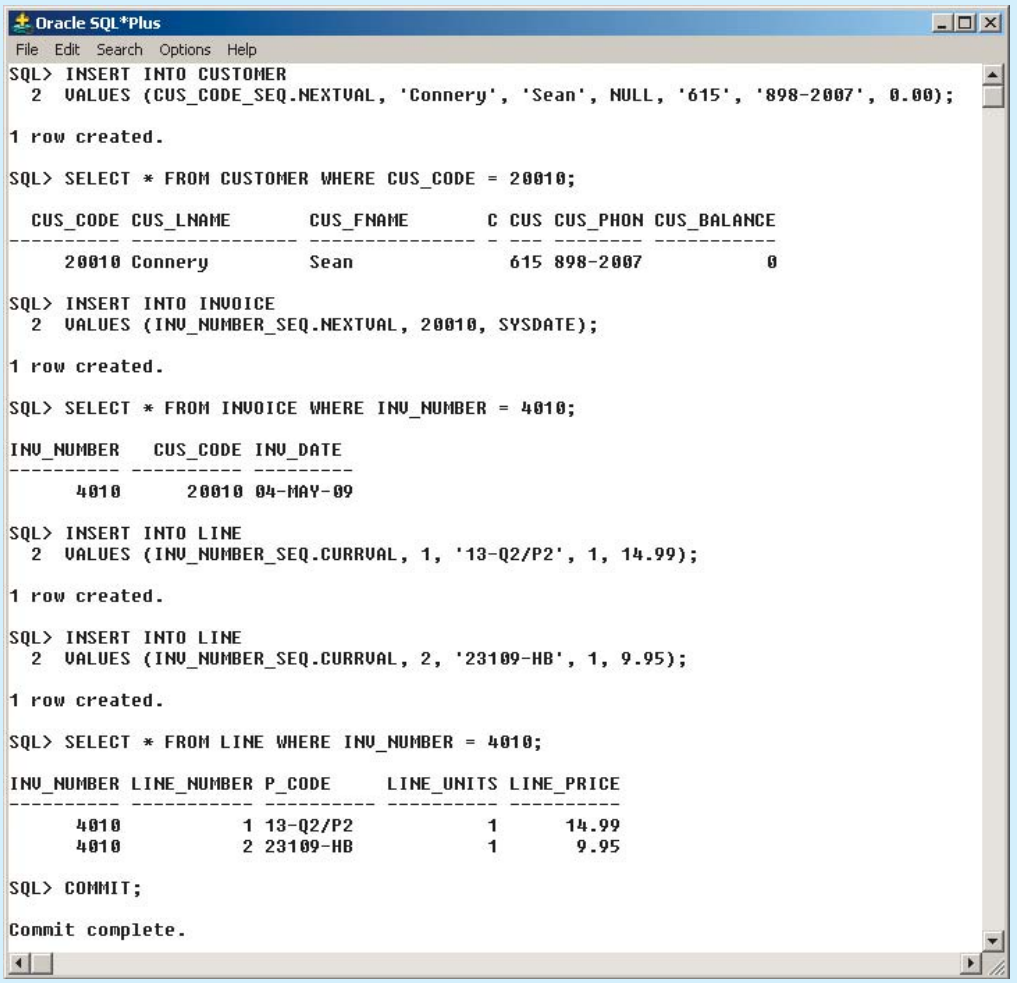
- CUS_CODE_SEQ.NEXTVAL retrieves the next available value from the sequence.
- Each time you use NEXTVAL, the sequence is incremented.
- Once a sequence value is used (through NEXTVAL), it cannot be used again. If, for some reason, your SQL statement rolls back, *the sequence value does not roll back*. If you issue another SQL statement (with another NEXTVAL), the next available sequence value will be returned to the user—it will look as though the sequence skips a number.
- You can issue an INSERT statement without using the sequence.

CURRVAL retrieves the current value of a sequence—that is, the last sequence number used, which was generated with a NEXTVAL. You cannot use CURRVAL unless a NEXTVAL was issued previously in the same session. The main use for CURRVAL is to enter rows in dependent tables. For example, the INVOICE and LINE tables are related in a one-to-many relationship through the INV_NUMBER attribute. You can use the INV_NUMBER_SEQ sequence to automatically generate invoice numbers. Then, using CURRVAL, you can get the latest INV_NUMBER used and assign it to the related INV_NUMBER foreign key attribute in the LINE table. For example:

```
INSERT INTO INVOICE VALUES (INV_NUMBER_SEQ.NEXTVAL, 20010, SYSDATE);
INSERT INTO LINE    VALUES (INV_NUMBER_SEQ.CURRVAL, 1,'13-Q2/P2', 1, 14.99);
INSERT INTO LINE    VALUES (INV_NUMBER_SEQ.CURRVAL, 2,'23109-HB', 1, 9.95);
COMMIT;
```

The results are shown in Figure 8.23.

**FIGURE 8.23** Oracle sequence examples




In the example shown in Figure 8.23, INV_NUMBER_SEQ.NEXTVAL retrieves the next available sequence number (4010) and assigns it to the INV_NUMBER column in the INVOICE table. Also note the use of the SYSDATE attribute to automatically insert the current date in the INV_DATE attribute. Next, the following two INSERT statements add the

products being sold to the LINE table. In this case, INV_NUMBER_SEQ.CURRVAL refers to the last-used INV_NUMBER_SEQ sequence number (4010). In this way, the relationship between INVOICE and LINE is established automatically. The COMMIT statement at the end of the command sequence makes the changes permanent. Of course, you can also issue a ROLLBACK statement, in which case the rows you inserted in INVOICE and LINE tables would be rolled back (but remember that the sequence number would not). Once you use a sequence number (with NEXTVAL), there is no way to reuse it! This "no-reuse" characteristic is designed to guarantee that the sequence will always generate unique values.

Remember these points when you think about sequences:

- The use of sequences is optional. You can enter the values manually.
- A sequence is not associated with a table. As in the examples in Figure 8.23, two distinct sequences were created (one for customer code values and one for invoice number values), but you could have created just one sequence and used it to generate unique values for both tables.

**NOTE**

The SQL-2003 standard defined the use of Identity columns and sequence objects. However, some DBMS vendors might not adhere to the standard. Check your DBMS documentation.

Finally, you can drop a sequence from a database with a DROP SEQUENCE command. For example, to drop the sequences created earlier, you would type:

```
DROP SEQUENCE CUS_CODE_SEQ;
DROP SEQUENCE INV_NUMBER_SEQ;
```

Dropping a sequence does not delete the values you assigned to table attributes (CUS_CODE and INV_NUMBER); it deletes only the sequence object from the database. The *values* you assigned to the table columns (CUS_CODE and INV_NUMBER) remain in the database.

Because the CUSTOMER and INVOICE tables are used in the following examples, you'll want to keep the original data set. Therefore, you can delete the customer, invoice, and line rows you just added by using the following commands:

```
DELETE FROM INVOICE WHERE INV_NUMBER = 4010;
DELETE FROM CUSTOMER WHERE CUS_CODE = 20010;
COMMIT;
```

Those commands delete the recently added invoice and all of the invoice line rows associated with the invoice (the LINE table's INV_NUMBER foreign key was defined with the ON DELETE CASCADE option) and the recently added customer. The COMMIT statement saves all changes to permanent storage.

**NOTE**

At this point, you'll need to re-create the CUS_CODE_SEQ and INV_NUMBER_SEQ sequences, as they will be used again later in the chapter. Enter:

```
CREATE SEQUENCE CUS_CODE_SEQ START WITH 20010 NOCACHE;
CREATE SEQUENCE INV_NUMBER_SEQ START WITH 4010 NOCACHE;
```

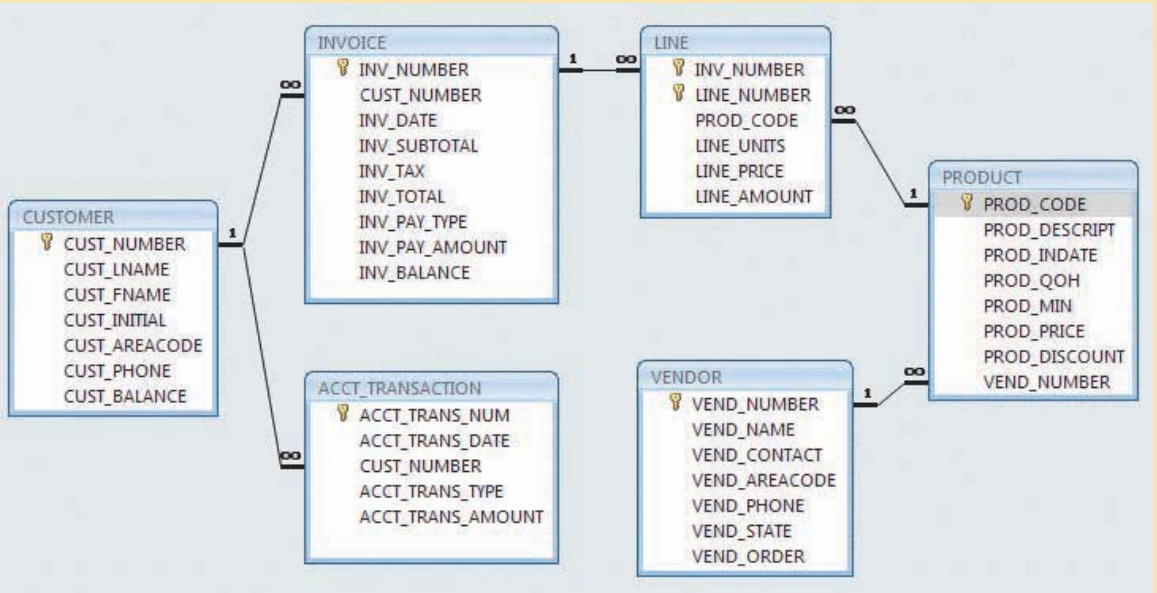
## 8.6 UPDATABLE VIEWS

In Chapter 7, you learned how to create a view and why and how views are used. You will now take a look at how views can be made to serve common data management tasks executed by database administrators.

## 10.1 WHAT IS A TRANSACTION?

To illustrate what transactions are and how they work, let's use the **Ch10_SaleCo** database. The relational diagram for that database is shown in Figure 10.1.

**FIGURE 10.1** The Ch10_SaleCo database relational diagram




### ONLINE CONTENT

The **Ch10_SaleCo** database used to illustrate the material in this chapter is found in the Premium Website for this book.

**NOTE**

Although SQL commands illustrate several transaction and concurrency control issues, you should be able to follow the discussions even if you have not studied **Chapter 7, Introduction to Structured Query Language (SQL)**, and **Chapter 8, Advanced SQL**. If you don't know SQL, ignore the SQL commands and focus on the discussions. If you have a working knowledge of SQL, you can use the **Ch10_SaleCo** database to generate your own SELECT and UPDATE examples and to augment the material presented in Chapters 7 and 8 by writing your own triggers and stored procedures.

As you examine the relational diagram in Figure 10.1, note the following features:

- The design stores the customer balance (CUST_BALANCE) value in the CUSTOMER table to indicate the total amount owed by the customer. The CUST_BALANCE attribute is increased when the customer makes a purchase on credit, and it is decreased when the customer makes a payment. Including the current customer account balance in the CUSTOMER table makes it very easy to write a query to determine the current balance for any customer and to generate important summaries such as total, average, minimum, and maximum balances.

- The ACCT_TRANSACTION table records all customer purchases and payments to track the details of customer account activity.

You could change the database design of the **Ch10_SaleCo** database to reflect accounting practice more precisely, but the implementation provided here will enable you to track the transactions well enough to serve the purpose of the chapter's discussions.

To understand the concept of a transaction, suppose that you sell a product to a customer. Furthermore, suppose that the customer may charge the purchase to his or her account. Given that scenario, your sales transaction consists of at least the following parts:

- You must write a new customer invoice.
- You must reduce the quantity on hand in the product's inventory.
- You must update the account transactions.
- You must update the customer balance.

The preceding sales transaction must be reflected in the database. In database terms, a **transaction** is any action that reads from and/or writes to a database. A transaction may consist of a simple SELECT statement to generate a list of table contents; it may consist of a series of related UPDATE statements to change the values of attributes in various tables; it may consist of a series of INSERT statements to add rows to one or more tables, or it may consist of a combination of SELECT, UPDATE, and INSERT statements. The sales transaction example includes a combination of INSERT and UPDATE statements.

Given the preceding discussion, you can now augment the definition of a transaction. A transaction is a *logical* unit of work that must be entirely completed or entirely aborted; no intermediate states are acceptable. In other words, a multicomponent transaction, such as the previously mentioned sale, must not be partially completed. Updating only the inventory or only the accounts receivable is not acceptable. All of the SQL statements in the transaction must be completed successfully. If any of the SQL statements fail, the entire transaction is rolled back to the original database state that existed before the transaction started. A successful transaction changes the database from one consistent state to another. A **consistent database state** is one in which all data integrity constraints are satisfied.

To ensure consistency of the database, every transaction must begin with the database in a known consistent state. If the database is not in a consistent state, the transaction will yield an inconsistent database that violates its integrity and business rules. For that reason, subject to limitations discussed later, all transactions are controlled and executed by the DBMS to guarantee database integrity.

Most real-world database transactions are formed by two or more database requests. A **database request** is the equivalent of a single SQL statement in an application program or transaction. For example, if a transaction is composed of two UPDATE statements and one INSERT statement, the transaction uses three database requests. In turn, each database request generates several input/output (I/O) operations that read from or write to physical storage media.

### 10.1.1 Evaluating Transaction Results

Not all transactions update the database. Suppose that you want to examine the CUSTOMER table to determine the current balance for customer number 10016. Such a transaction can be completed by using the SQL code:

```
SELECT      CUST_NUMBER, CUST_BALANCE
FROM        CUSTOMER
WHERE       CUST_NUMBER = 10016;
```

Although that query does not make any changes in the CUSTOMER table, the SQL code represents a transaction because it *accesses* the database. If the database existed in a consistent state before the access, the database remains in a consistent state after the access because the transaction did not alter the database.

Remember that a transaction may consist of a single SQL statement or a collection of related SQL statements. Let's revisit the previous sales example to illustrate a more complex transaction, using the **Ch10_SaleCo** database. Suppose that on January 18, 2010 you register the credit sale of one unit of product 89-WRE-Q to customer 10016 in the amount of $277.55. The required transaction affects the INVOICE, LINE, PRODUCT, CUSTOMER, and ACCT_TRANSACTION tables. The SQL statements that represent this transaction are as follows:

```
INSERT INTO INVOICE
      VALUES (1009, 10016,'18-Jan-2010', 256.99, 20.56, 277.55, 'cred', 0.00, 277.55);
INSERT INTO LINE
      VALUES (1009, 1, '89-WRE-Q', 1, 256.99, 256.99);

UPDATE PRODUCT
SET PROD_QOH = PROD_QOH – 1
WHERE PROD_CODE = '89-WRE-Q';

UPDATE CUSTOMER
SET CUST_BALANCE = CUST_BALANCE + 277.55
WHERE CUST_NUMBER = 10016;

INSERT INTO ACCT_TRANSACTION
      VALUES (10007, '18-Jan-10', 10016, 'charge', 277.55);
COMMIT;
```

The results of the successfully completed transaction are shown in Figure 10.2. (Note that all records involved in the transaction have been outlined in red.)

To further your understanding of the transaction results, note the following:

- A new row 1009 was added to the INVOICE table. In this row, derived attribute values were stored for the invoice subtotal, the tax, the invoice total, and the invoice balance.
- The LINE row for invoice 1009 was added to reflect the purchase of one unit of product 89-WRE-Q with a price of $256.99. In this row, the derived attribute values for the line amount were stored.
- The product 89-WRE-Q's quantity on hand (PROD_QOH) in the PRODUCT table was reduced by one (the initial value was 12), thus leaving a quantity on hand of 11.
- The customer balance (CUST_BALANCE) for customer 10016 was updated by adding $277.55 to the existing balance (the initial value was $0.00).
- A new row was added to the ACCT_TRANSACTION table to reflect the new account transaction number 10007.
- The COMMIT statement is used to end a successful transaction. (See Section 10.1.3.)

Now suppose that the DBMS completes the first three SQL statements. Furthermore, suppose that during the execution of the fourth statement (the UPDATE of the CUSTOMER table's CUST_BALANCE value for customer 10016), the computer system experiences a loss of electrical power. If the computer does not have a backup power supply, the transaction cannot be completed. Therefore, the INVOICE and LINE rows were added, the PRODUCT table was updated to represent the sale of product 89-WRE-Q, but customer 10016 was not charged, nor was the required record in the ACCT_TRANSACTION table written. The database is now in an inconsistent state, and it is not usable for subsequent transactions. Assuming that the DBMS supports transaction management, *the DBMS will roll back the database to a previous consistent state*.

**FIGURE 10.2** Tracing the transaction in the Ch10_SaleCo database

**Database name: ch10_SaleCo**

**Table name: INVOICE**

| INV_NUMBER | CUST_NUMBER | INV_DATE | INV_SUBTOTAL | INV_TAX | INV_TOTAL | INV_PAY_TYPE | INV_PAY_AMOUNT | INV_BALANCE |
|---|---|---|---|---|---|---|---|---|
| 1001 | 10014 | 16-Jan-10 | 54.92 | 4.39 | 59.31 | cc | 59.31 | 0.00 |
| 1002 | 10011 | 16-Jan-10 | 9.96 | 0.80 | 10.76 | cash | 10.76 | 0.00 |
| 1003 | 10012 | 16-Jan-10 | 270.70 | 21.66 | 292.36 | cc | 292.36 | 0.00 |
| 1004 | 10011 | 17-Jan-10 | 34.87 | 2.79 | 37.66 | cc | 37.66 | 0.00 |
| 1005 | 10018 | 17-Jan-10 | 70.44 | 5.64 | 76.08 | cc | 76.08 | 0.00 |
| 1006 | 10014 | 17-Jan-10 | 397.83 | 31.83 | 429.66 | cred | 100.00 | 329.66 |
| 1007 | 10015 | 17-Jan-10 | 34.97 | 2.80 | 37.77 | chk | 37.77 | 0.00 |
| 1008 | 10011 | 17-Jan-10 | 1033.08 | 82.65 | 1115.73 | cred | 500.00 | 615.73 |
| 1009 | 10016 | 18-Jan-10 | 256.99 | 20.56 | 277.55 | cred | 0.00 | 277.55 |

**Table name: LINE**

| INV_NUMBER | LINE_NUMBER | PROD_CODE | LINE_UNITS | LINE_PRICE | LINE_AMOUNT |
|---|---|---|---|---|---|
| 1001 | 1 | 13-Q2/P2 | 3 | 14.99 | 44.97 |
| 1001 | 2 | 23109-HB | 1 | 9.95 | 9.95 |
| 1002 | 1 | 54778-2T | 2 | 4.99 | 9.98 |
| 1003 | 1 | 2238/QPD | 4 | 38.95 | 155.80 |
| 1003 | 2 | 1546-QQ2 | 1 | 39.95 | 39.95 |
| 1003 | 3 | 13-Q2/P2 | 5 | 14.99 | 74.95 |
| 1004 | 1 | 54778-2T | 3 | 4.99 | 14.97 |
| 1004 | 2 | 23109-HB | 2 | 9.95 | 19.90 |
| 1005 | 1 | PVC23DRT | 12 | 5.87 | 70.44 |
| 1006 | 1 | SM-18277 | 3 | 6.99 | 20.97 |
| 1006 | 2 | 2232/QTY | 1 | 109.92 | 109.92 |
| 1006 | 3 | 23109-HB | 1 | 9.95 | 9.95 |
| 1006 | 4 | 89-WRE-Q | 1 | 256.99 | 256.99 |
| 1007 | 1 | 13-Q2/P2 | 2 | 14.99 | 29.98 |
| 1007 | 2 | 54778-2T | 1 | 4.99 | 4.99 |
| 1008 | 1 | PVC23DRT | 5 | 5.87 | 29.35 |
| 1008 | 2 | WR3/TT3 | 4 | 119.95 | 479.80 |
| 1008 | 3 | 23109-HB | 1 | 9.95 | 9.95 |
| 1008 | 4 | 89-WRE-Q | 2 | 256.99 | 513.98 |
| 1009 | 1 | 89-WRE-Q | 1 | 256.99 | 256.99 |

**Table name: PRODUCT**

| PROD_CODE | PROD_DESCRIPT | PROD_INDATE | PROD_QOH | PROD_MIN | PROD_PRICE | PROD_DISCOUNT | VEND_NUMBER |
|---|---|---|---|---|---|---|---|
| 11QER/31 | Power painter, 15 psi., 3-nozzle | 03-Nov-09 | 8 | 5 | 109.99 | 0.00 | 25595 |
| 13-Q2/P2 | 7.25-in. pwr. saw blade | 13-Dec-09 | 32 | 15 | 14.99 | 0.05 | 21344 |
| 14-Q1/L3 | 9.00-in. pwr. saw blade | 13-Nov-09 | 18 | 12 | 17.49 | 0.00 | 21344 |
| 1546-QQ2 | Hrd. cloth, 1/4-in., 2x50 | 15-Jan-10 | 15 | 8 | 39.95 | 0.00 | 23119 |
| 1558-QW1 | Hrd. cloth, 1/2-in., 3x50 | 15-Jan-10 | 23 | 5 | 43.99 | 0.00 | 23119 |
| 2232/QTY | B&D jigsaw, 12-in. blade | 30-Dec-09 | 8 | 5 | 109.92 | 0.05 | 24288 |
| 2232/QWE | B&D jigsaw, 8-in. blade | 24-Dec-09 | 6 | 5 | 99.87 | 0.05 | 24288 |
| 2238/QPD | B&D cordless drill, 1/2-in. | 20-Jan-10 | 12 | 5 | 38.95 | 0.05 | 25595 |
| 23109-HB | Claw hammer | 20-Jan-10 | 23 | 10 | 9.95 | 0.10 | 21225 |
| 23114-AA | Sledge hammer, 12 lb. | 02-Jan-10 | 8 | 5 | 14.40 | 0.05 | |
| 54778-2T | Rat-tail file, 1/8-in. fine | 15-Dec-09 | 43 | 20 | 4.99 | 0.00 | 21344 |
| 89-WRE-Q | Hicut chain saw, 16 in. | 07-Jan-10 | 11 | 5 | 256.99 | 0.05 | 24288 |
| PVC23DRT | PVC pipe, 3.5-in., 8-ft | 06-Jan-10 | 188 | 75 | 5.87 | 0.00 | |
| SM-18277 | 1.25-in. metal screw, 25 | 01-Mar-10 | 172 | 75 | 6.99 | 0.00 | 21225 |
| SW-23116 | 2.5-in. wd. screw, 50 | 24-Feb-10 | 237 | 100 | 8.45 | 0.00 | 21231 |
| WR3/TT3 | Steel matting, 4'x8'x1/6", .5" mesh | 17-Jan-10 | 18 | 5 | 119.95 | 0.10 | 25595 |

**Table name: CUSTOMER**

| CUST_NUMB | CUST_LNAME | CUST_FNAME | CUST_INITIAL | CUST_AREACODE | CUST_PHONE | CUST_BALANCE |
|---|---|---|---|---|---|---|
| 10010 | Ramas | Alfred | A | 615 | 844-2573 | 0.00 |
| 10011 | Dunne | Leona | K | 713 | 894-1238 | 615.73 |
| 10012 | Smith | Kathy | W | 615 | 894-2285 | 0.00 |
| 10013 | Olowski | Paul | F | 615 | 894-2180 | 0.00 |
| 10014 | Orlando | Myron | | 615 | 222-1672 | 0.00 |
| 10015 | O'Brian | Amy | B | 713 | 442-3381 | 0.00 |
| 10016 | Brown | James | G | 615 | 297-1228 | 277.55 |
| 10017 | Williams | George | | 615 | 290-2556 | 0.00 |
| 10018 | Farriss | Anne | G | 713 | 382-7185 | 0.00 |
| 10019 | Smith | Olette | K | 615 | 297-3809 | 0.00 |

**Table name: ACCT_TRANSACTION**

| ACCT_TRANS_NUM | ACCT_TRANS_DATE | CUST_NUMBER | ACCT_TRANS_TYPE | ACCT_TRANS_AMOUNT |
|---|---|---|---|---|
| 10003 | 17-Jan-10 | 10014 | charge | 329.66 |
| 10004 | 17-Jan-10 | 10011 | charge | 615.73 |
| 10006 | 29-Jan-10 | 10014 | payment | 329.66 |
| 10007 | 18-Jan-10 | 10016 | charge | 277.55 |

**NOTE**

By default, MS Access does not support transaction management as discussed here. More sophisticated DBMSs, such as Oracle, SQL Server, and DB2, do support the transaction management components discussed in this chapter.

Although the DBMS is designed to recover a database to a previous consistent state when an interruption prevents the completion of a transaction, the transaction itself is defined by the end user or programmer and must be semantically correct. *The DBMS cannot guarantee that the semantic meaning of the transaction truly represents the real-world event.* For example, suppose that following the sale of 10 units of product 89-WRE-Q, the inventory UPDATE commands were written this way:

```
UPDATE PRODUCT
SET PROD_QOH = PROD_QOH + 10
WHERE PROD_CODE = '89-WRE-Q';
```

The sale should have *decreased* the PROD_QOH value for product 89-WRE-Q by 10. Instead, the UPDATE *added* 10 to product 89-WRE-Q's PROD_QOH value.

Although the UPDATE command's syntax is correct, its use yields incorrect results. Yet the DBMS will execute the transaction anyway. The DBMS cannot evaluate whether the transaction represents the real-world event correctly; that is the end user's responsibility. End users and programmers are capable of introducing many errors in this fashion.

Imagine the consequences of reducing the quantity on hand for product 1546-QQ2 instead of product 89-WRE-Q or of crediting the CUST_BALANCE value for customer 10012 rather than customer 10016.

Clearly, improper or incomplete transactions can have a devastating effect on database integrity. Some DBMSs—*especially* the relational variety—provide means by which the user can define enforceable constraints based on business rules. Other integrity rules, such as those governing referential and entity integrity, are enforced automatically by the DBMS when the table structures are properly defined, thereby letting the DBMS validate some transactions. For example, if a transaction inserts a new customer number into a customer table and the customer number being inserted already exists, the DBMS will end the transaction with an error code to indicate a violation of the primary key integrity rule.

### 10.1.2 Transaction Properties

Each individual transaction must display *atomicity*, *consistency*, *isolation*, and *durability*. These properties are sometimes referred to as the ACID test. In addition, when executing multiple transactions, the DBMS must schedule the concurrent execution of the transaction's operations. The schedule of such transaction's operations must exhibit the property of *serializability*. Let's look briefly at each of the properties.

- **Atomicity** requires that *all* operations (SQL requests) of a transaction be completed; if not, the transaction is aborted. If a transaction T1 has four SQL requests, all four requests must be successfully completed; otherwise, the entire transaction is aborted. In other words, a transaction is treated as a single, indivisible, logical unit of work.
- **Consistency** indicates the permanence of the database's consistent state. A transaction takes a database from one consistent state to another consistent state. When a transaction is completed, the database must be in a consistent state; if any of the transaction parts violates an integrity constraint, the entire transaction is aborted.
- **Isolation** means that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed. In other words, if a transaction T1 is being executed and is using the data item X, that data item cannot be accessed by any other transaction (T2 ... T*n*) until T1 ends. This property is particularly useful in multiuser database environments because several users can access and update the database at the same time.
- **Durability** ensures that once transaction changes are done (committed), they cannot be undone or lost, even in the event of a system failure.
- **Serializability** ensures that the schedule for the concurrent execution of the transactions yields consistent results. This property is important in multiuser and distributed databases, where multiple transactions are likely to be executed concurrently. Naturally, if only a single transaction is executed, serializability is not an issue.

A single-user database system automatically ensures serializability and isolation of the database because only one transaction is executed at a time. The atomicity, consistency, and durability of transactions must be guaranteed by the single-user DBMSs. (Even a single-user DBMS must manage recovery from errors created by operating-system-induced interruptions, power interruptions, and improper application execution.)

Multiuser databases are typically subject to multiple concurrent transactions. Therefore, the multiuser DBMS must implement controls to ensure serializability and isolation of transactions—in addition to atomicity and durability—to guard the database's consistency and integrity. For example, if several concurrent transactions are executed over the same data set and the second transaction updates the database before the first transaction is finished, the isolation property is violated and the database is no longer consistent. The DBMS must manage the transactions by using concurrency control techniques to avoid such undesirable situations.

### 10.1.3 Transaction Management with SQL

The American National Standards Institute (ANSI) has defined standards that govern SQL database transactions. Transaction support is provided by two SQL statements: COMMIT and ROLLBACK. The ANSI standards require that

TABLE 10.15 A Transaction Log for Transaction Recovery Examples

| TRL ID | TRX NUM | PREV PTR | NEXT PTR | OPERATION | TABLE | ROW ID | ATTRIBUTE | BEFORE VALUE | AFTER VALUE |
|---|---|---|---|---|---|---|---|---|---|
| 341 | 101 | Null | 352 | START | ****Start Transaction | | | | |
| 352 | 101 | 341 | 363 | UPDATE | PRODUCT | 54778-2T | PROD_QOH | 45 | 43 |
| 363 | 101 | 352 | 365 | UPDATE | CUSTOMER | 10011 | CUST_BALANCE | 615.73 | 675.62 |
| 365 | 101 | 363 | Null | COMMIT | **** End of Transaction | | | | |
| 397 | 106 | Null | 405 | START | ****Start Transaction | | | | |
| 405 | 106 | 397 | 415 | INSERT | INVOICE | 1009 | | | 1009,10016, ... |
| 415 | 106 | 405 | 419 | INSERT | LINE | 1009,1 | | | 1009,1, 89-WRE-Q,1, ... |
| 419 | 106 | 415 | 427 | UPDATE | PRODUCT | 89-WRE-Q | PROD_QOH | 12 | 11 |
| 423 | | | | CHECKPOINT | | | | | |
| 427 | 106 | 419 | 431 | UPDATE | CUSTOMER | 10016 | CUST_BALANCE | 0.00 | 277.55 |
| 431 | 106 | 427 | 457 | INSERT | ACCT_TRANSACTION | 10007 | | | 1007,18-JAN-2010, ... |
| 457 | 106 | 431 | Null | COMMIT | **** End of Transaction | | | | |
| 521 | 155 | Null | 525 | START | ****Start Transaction | | | | |
| 525 | 155 | 521 | 528 | UPDATE | PRODUCT | 2232/QWE | PROD_QOH | 6 | 26 |
| 528 | 155 | 525 | Null | COMMIT | **** End of Transaction | | | | |
| * * * * * C *R*A* S* H * * * * | | | | | | | | | |

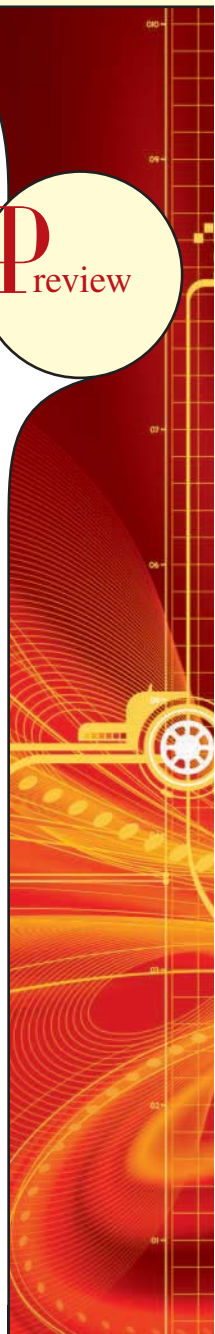
12

TWELVE

# Distributed Database Management Systems

**In this chapter, you will learn:**

- What a distributed database management system (DDBMS) is and what its components are
- How database implementation is affected by different levels of data and process distribution
- How transactions are managed in a distributed database environment
- How database design is affected by the distributed database environment



## Preview

In this chapter, you will learn that a single database can be divided into several fragments. The fragments can be stored on different computers within a network. Processing, too, can be dispersed among several different network sites, or nodes. The multisite database forms the core of the distributed database system.

The growth of distributed database systems has been fostered by the dispersion of business operations across the country and around the world, along with the rapid pace of technological change that has made local and wide area networks practical and more reliable. The network-based distributed database system is very flexible: it can serve the needs of a small business operating two stores in the same town while at the same time meeting the needs of a global business.

Although a distributed database system requires a more sophisticated DBMS, the end user should not be burdened by increased operational complexity. That is, the greater complexity of a distributed database system should be transparent to the end user.

The distributed database management system (DDBMS) treats a distributed database as a single logical database; therefore, the basic design concepts you learned in earlier chapters apply. However, although the end user need not be aware of the distributed database's special characteristics, the distribution of data among different sites in a computer network clearly adds to a system's complexity. For example, the design of a distributed database must consider the location of the data and the partitioning of the data into database fragments. You will examine such issues in this chapter.

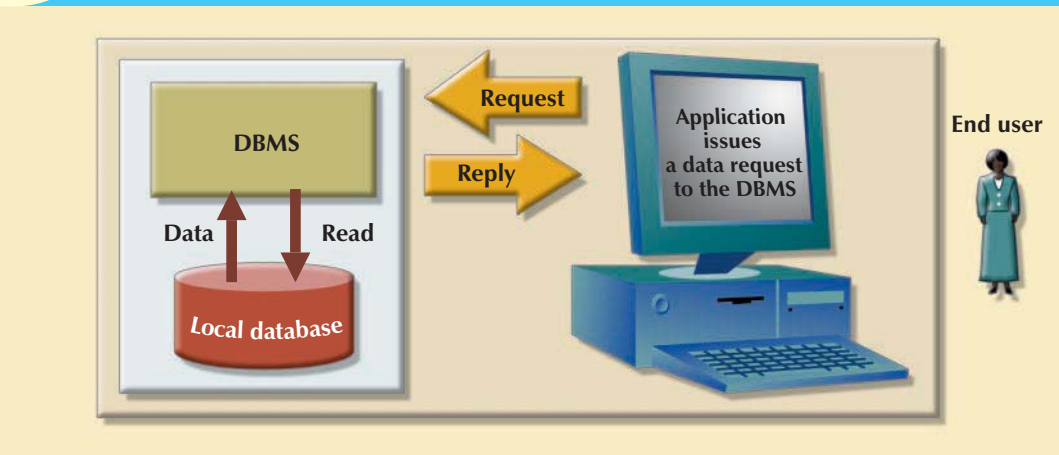
## 12.1 THE EVOLUTION OF DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

A **distributed database management system** (**DDBMS**) governs the storage and processing of logically related data over interconnected computer systems in which both data and processing are distributed among several sites. To understand how and why the DDBMS is different from the DBMS, it is useful to briefly examine the changes in the business environment that set the stage for the development of the DDBMS.

During the 1970s, corporations implemented centralized database management systems to meet their structured information needs. Structured information is usually presented as regularly issued formal reports in a standard format. Such information, generated by procedural programming languages, is created by specialists in response to precisely channeled requests. Thus, structured information needs are well served by centralized systems.

The use of a centralized database required that corporate data be stored in a single central site, usually a mainframe computer. Data access was provided through dumb terminals. The centralized approach, illustrated in Figure 12.1, worked well to fill the structured information needs of corporations, but it fell short when quickly moving events required faster response times and equally quick access to information. The slow progression from information request to approval to specialist to user simply did not serve decision makers well in a dynamic environment. What was needed was quick, unstructured access to databases, using ad hoc queries to generate on-the-spot information.

**FIGURE 12.1** Centralized database management system




Database management systems based on the relational model could provide the environment in which unstructured information needs would be met by employing ad hoc queries. End users would be given the ability to access data when needed. Unfortunately, the early relational model implementations did not yet deliver acceptable throughput when compared to the well-established hierarchical or network database models.

The last two decades gave birth to a series of crucial social and technological changes that affected database development and design. Among those changes were:

- Business operations became decentralized.
- Competition increased at the global level.
- Customer demands and market needs favored a decentralized management style.
- Rapid technological change created low-cost computers with mainframe-like power, impressive multifunction handheld portable wireless devices with cellular phone and data services, and increasingly complex and fast networks to connect them. As a consequence, corporations have increasingly adopted advanced network technologies as the platform for their computerized solutions.

- The large number of applications based on DBMSs and the need to protect investments in centralized DBMS software made the notion of data sharing attractive. Data realms are converging in the digital world more and more. As a result, single applications manage multiple different types of data (voice, video, music, images, etc.), and such data are accessed from multiple geographically dispersed locations.

Those factors created a dynamic business environment in which companies had to respond quickly to competitive and technological pressures. As large business units restructured to form leaner, quickly reacting, dispersed operations, two database requirements became obvious:

- *Rapid ad hoc data access* became crucial in the quick-response decision-making environment.
- *The decentralization of management structures* based on the decentralization of business units made decentralized multiple-access and multiple-location databases a necessity.

During recent years, the factors just described became even more firmly entrenched. However, the way those factors were addressed was strongly influenced by:

- *The growing acceptance of the Internet as the platform for data access and distribution*. The World Wide Web (the Web) is, in effect, the *repository* for distributed data.
- *The wireless revolution*. The widespread use of wireless digital devices, such as smart phones like the iPhone and BlackBerry and personal digital assistants (PDAs), has created high demand for data access. Such devices access data from geographically dispersed locations and require varied data exchanges in multiple formats (data, voice, video, music, pictures, etc.) Although distributed data access does not necessarily imply distributed databases, performance and failure tolerance requirements often make use of data replication techniques similar to the ones found in distributed databases.
- *The accelerated growth of companies providing "application as a service" type of services*. This new type of service provides remote application services to companies wanting to outsource their application development, maintenance, and operations. The company data is generally stored on central servers and is not necessarily distributed. Just as with wireless data access, this type of service may not require fully distributed data functionality; however, other factors such as performance and failure tolerance often require the use of data replication techniques similar to the ones found in distributed databases.
- *The increased focus on data analysis that led to data mining and data warehousing*. Although a data warehouse is not usually a distributed database, it does rely on techniques such as data replication and distributed queries that facilitate data extraction and integration. (Data warehouse design, implementation, and use are discussed in Chapter 13, Business Intelligence and Data Warehouses.)

**ONLINE CONTENT**

To learn more about the Internet's impact on data access and distribution, see **Appendix I, Databases in Electronic Commerce,** in the Premium Website for this book**.**

At this point, the long-term impact of the Internet and the wireless revolution on *distributed* database design and management is still unclear. Perhaps the success of the Internet and wireless technologies will foster the use of distributed databases as bandwidth becomes a more troublesome bottleneck. Perhaps the resolution of bandwidth problems will simply confirm the centralized database standard. In any case, distributed databases exist today and many distributed database operating concepts and components are likely to find a place in future database developments.

The decentralized database is especially desirable because centralized database management is subject to problems such as:

- *Performance degradation* because of a growing number of remote locations over greater distances.
- *High costs* associated with maintaining and operating large central (mainframe) database systems.

- *Reliability problems* created by dependence on a central site (single point of failure syndrome) and the need for data replication.
- *Scalability problems* associated with the physical limits imposed by a single location (power, temperature conditioning, and power consumption.)
- *Organizational rigidity* imposed by the database might not support the flexibility and agility required by modern global organizations.

The dynamic business environment and the centralized database's shortcomings spawned a demand for applications based on accessing data from different sources at multiple locations. Such a multiple-source/multiple-location database environment is best managed by a distributed database management system (DDBMS).

## 12.2 DDBMS ADVANTAGES AND DISADVANTAGES

Distributed database management systems deliver several advantages over traditional systems. At the same time, they are subject to some problems. Table 12.1 summarizes the advantages and disadvantages associated with a DDBMS.

**TABLE 12.1 Distributed DBMS Advantages and Disadvantages**

| ADVANTAGES | DISADVANTAGES |
|---|---|
| • *Data are located near the greatest demand site*. The data in a distributed database system are dispersed to match business requirements.<br>• *Faster data access*. End users often work with only a locally stored subset of the company's data.<br>• *Faster data processing*. A distributed database system spreads out the systems workload by processing data at several sites.<br>• *Growth facilitation*. New sites can be added to the network without affecting the operations of other sites.<br>• *Improved communications*. Because local sites are smaller and located closer to customers, local sites foster better communication among departments and between customers and company staff.<br>• *Reduced operating costs*. It is more cost-effective to add workstations to a network than to update a mainframe system. Development work is done more cheaply and more quickly on low-cost PCs than on mainframes.<br>• *User-friendly interface*. PCs and workstations are usually equipped with an easy-to-use graphical user interface (GUI). The GUI simplifies training and use for end users.<br>• *Less danger of a single-point failure*. When one of the computers fails, the workload is picked up by other workstations. Data are also distributed at multiple sites.<br>• *Processor independence*. The end user is able to access any available copy of the data, and an end user's request is processed by any processor at the data location. | • *Complexity of management and control*. Applications must recognize data location, and they must be able to stitch together data from various sites. Database administrators must have the ability to coordinate database activities to prevent database degradation due to data anomalies.<br>• *Technological difficulty*. Data integrity, transaction management, concurrency control, security, backup, recovery, query optimization, access path selection, and so on, must all be addressed and resolved.<br>• *Security*. The probability of security lapses increases when data are located at multiple sites. The responsibility of data management will be shared by different people at several sites.<br>• *Lack of standards*. There are no standard communication protocols at the database level. (Although TCP/IP is the de facto standard at the network level, there is no standard at the application level.) For example, different database vendors employ different—and often incompatible—techniques to manage the distribution of data and processing in a DDBMS environment.<br>• *Increased storage and infrastructure requirements*. Multiple copies of data are required at different sites, thus requiring additional disk storage space.<br>• *Increased training cost*. Training costs are generally higher in a distributed model than they would be in a centralized model, sometimes even to the extent of offsetting operational and hardware savings.<br>• *Costs*. Distributed databases require duplicated infrastructure to operate (physical location, environment, personnel, software, licensing, etc.) |

Distributed databases are used successfully but have a long way to go before they will yield the full flexibility and power of which they are theoretically capable. The inherently complex distributed data environment increases the urgency for standard protocols governing transaction management, concurrency control, security, backup, recovery, query optimization, access path selection, and so on. Such issues must be addressed and resolved before DDBMS technology is widely embraced.

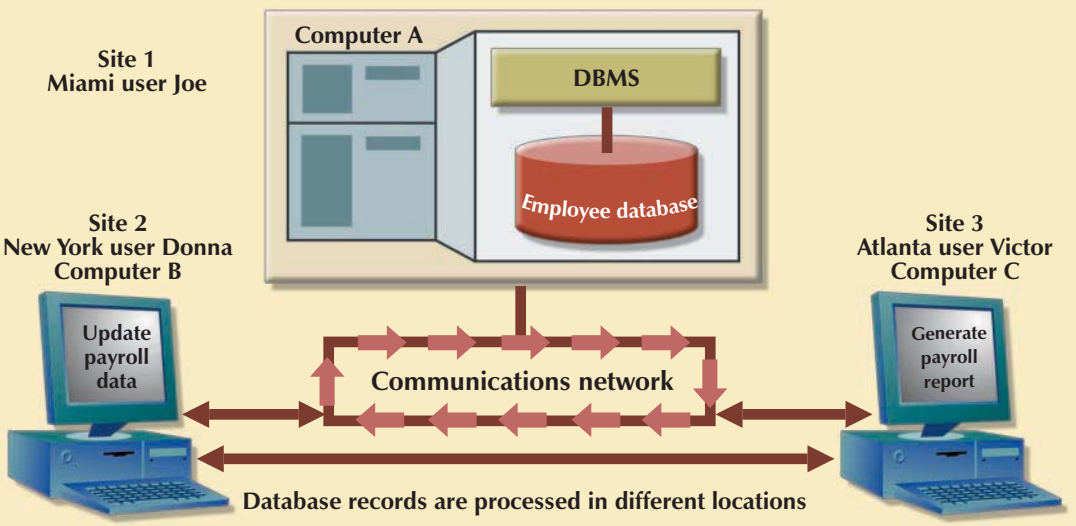
The remainder of this chapter explores the basic components and concepts of the distributed database. Because the distributed database is usually based on the relational database model, relational terminology is used to explain the basic concepts and components of a distributed database.

## 12.3 DISTRIBUTED PROCESSING AND DISTRIBUTED DATABASES

In **distributed processing**, a database's logical processing is shared among two or more physically independent sites that are connected through a network. For example, the data input/output (I/O), data selection, and data validation might be performed on one computer, and a report based on that data might be created on another computer.

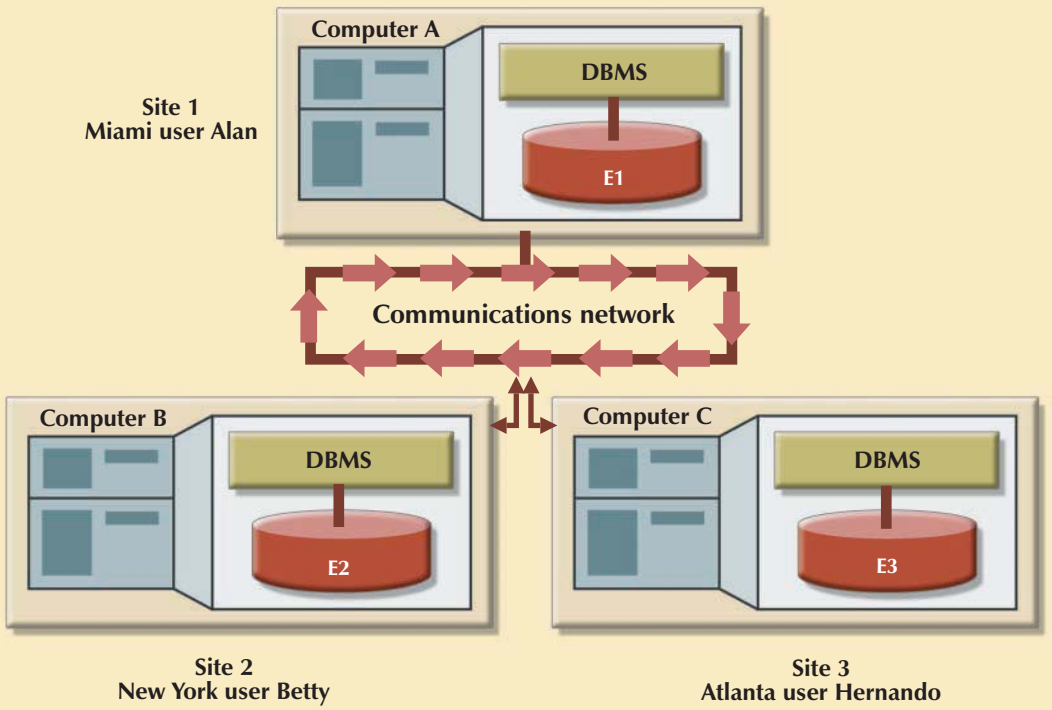
A basic distributed processing environment is illustrated in Figure 12.2, which shows that a distributed processing system shares the database processing chores among three sites connected through a communications network. Although the database resides at only one site (Miami), each site can access the data and update the database. The database is located on Computer A, a network computer known as the *database server*.

**FIGURE 12.2** **Distributed processing environment**




A **distributed database**, on the other hand, stores a logically related database over two or more physically independent sites. The sites are connected via a computer network. In contrast, the distributed processing system uses only a single-site database but shares the processing chores among several sites. In a distributed database system, a database is composed of several parts known as **database fragments**. The database fragments are located at different sites and can be replicated among various sites. Each database fragment is, in turn, managed by its local database process. An example of a distributed database environment is shown in Figure 12.3.

**FIGURE 12.3** Distributed database environment




The database in Figure 12.3 is divided into three database fragments (E1, E2, and E3) located at different sites. The computers are connected through a network system. In a fully distributed database, the users Alan, Betty, and Hernando do not need to know the name or location of each database fragment in order to access the database. Also, the users might be located at sites other than Miami, New York, or Atlanta and still be able to access the database as a single logical unit.

As you examine Figures 12.2 and 12.3, you should keep the following points in mind:

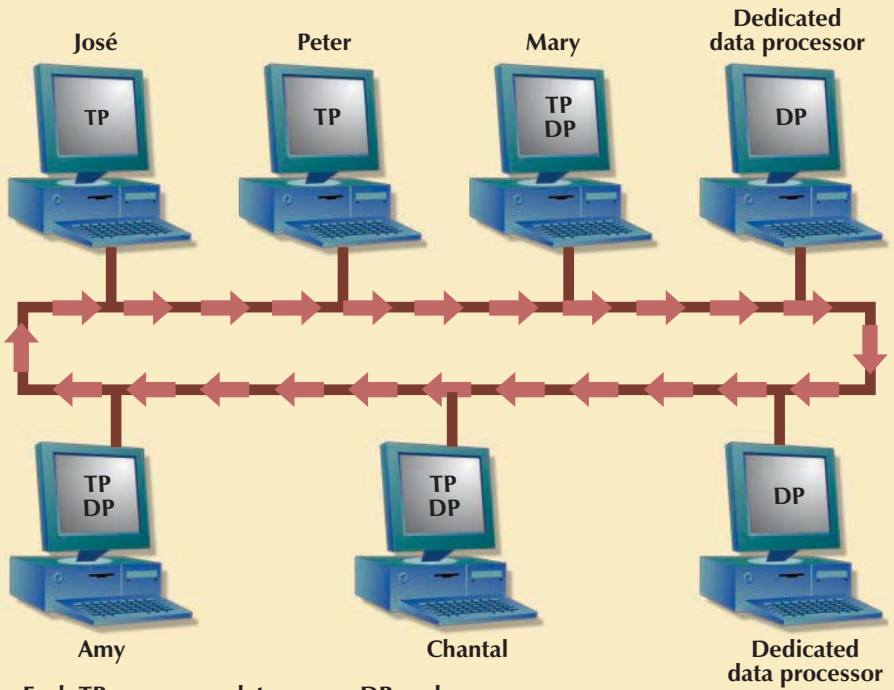
- Distributed processing does not require a distributed database, but a distributed database requires distributed processing (each database fragment is managed by its own local database process).
- Distributed processing may be based on a single database located on a single computer. For the management of distributed data to occur, copies or parts of the database processing functions must be distributed to all data storage sites.
- Both distributed processing and distributed databases require a network to connect all components.

## 12.4 CHARACTERISTICS OF DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

A DDBMS governs the storage and processing of logically related data over interconnected computer systems in which both data and processing functions are distributed among several sites. A DBMS must have at least the following functions to be classified as distributed:

- *Application interface* to interact with the end user, application programs, and other DBMSs within the distributed database.
- *Validation* to analyze data requests for syntax correctness.
- *Transformation* to decompose complex requests into atomic data request components.

**FIGURE 12.5** Distributed database system management components




*Note*: Each TP can access data on any DP, and each DP handles all requests for local data from any TP.

## 12.6 LEVELS OF DATA AND PROCESS DISTRIBUTION

Current database systems can be classified on the basis of how process distribution and data distribution are supported. For example, a DBMS may store data in a single site (centralized DB) or in multiple sites (distributed DB) and may support data processing at a single site or at multiple sites. Table 12.2 uses a simple matrix to classify database systems according to data and process distribution. These types of processes are discussed in the sections that follow.

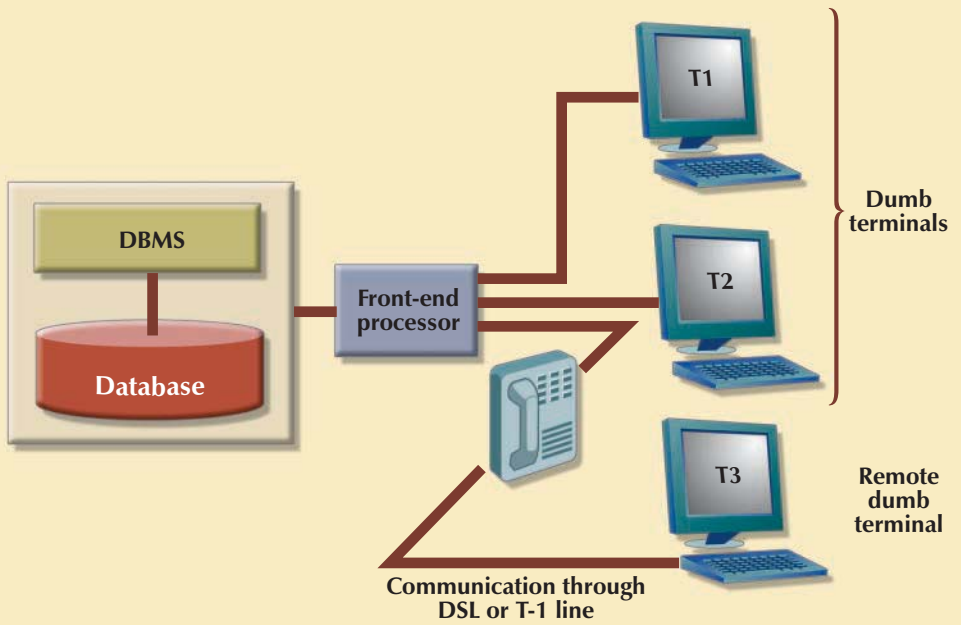
**TABLE 12.2** Database Systems: Levels of Data and Process Distribution

| | SINGLE-SITE DATA | MULTIPLE-SITE DATA |
|---|---|---|
| **Single-site process** | Host DBMS | Not applicable<br>(Requires multiple processes) |
| **Multiple-site process** | File server<br>Client/server DBMS (LAN DBMS) | Fully distributed<br>Client/server DDBMS |

### 12.6.1 Single-Site Processing, Single-Site Data (SPSD)

In the **single-site processing**, **single-site data** (**SPSD**) scenario, all processing is done on a single host computer (single-processor server, multiprocessor server, mainframe system) and all data are stored on the host computer's local disk system. Processing cannot be done on the end user's side of the system. Such a scenario is typical of most mainframe and midrange server computer DBMSs. The DBMS is located on the host computer, which is accessed by dumb terminals connected to it. (See Figure 12.6.) This scenario is also typical of the first generation of single-user microcomputer databases.

**FIGURE 12.6** Single-site processing, single-site data (centralized)




Using Figure 12.6 as an example, you can see that the functions of the TP and the DP are embedded within the DBMS located on a single computer. The DBMS usually runs under a time-sharing, multitasking operating system, which allows several processes to run concurrently on a host computer accessing a single DP. All data storage and data processing are handled by a single host computer.

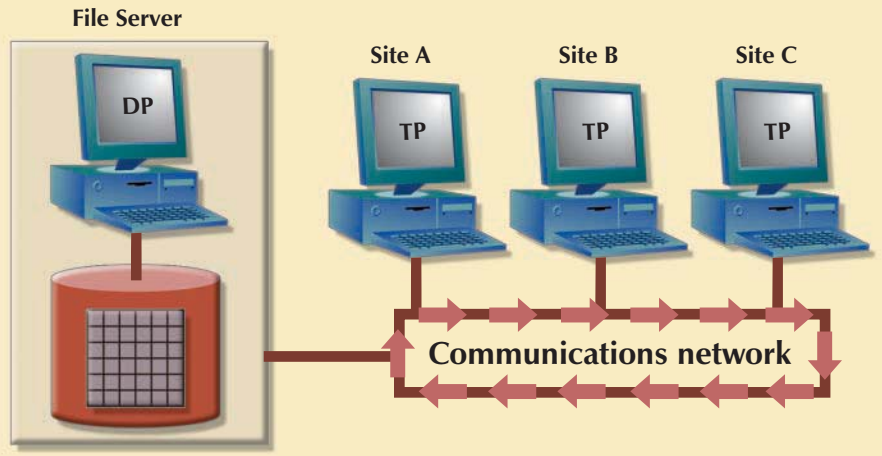
### 12.6.2 MULTIPLE-SITE PROCESSING, SINGLE-SITE DATA (MPSD)

Under the **multiple-site processing**, **single-site data** (**MPSD**) scenario, multiple processes run on different computers sharing a single data repository. Typically, the MPSD scenario requires a network file server running conventional applications that are accessed through a network. Many multiuser accounting applications running under a personal computer network fit such a description. (See Figure 12.7.)

As you examine Figure 12.7, Note that:

- The TP on each workstation acts only as a redirector to route all network data requests to the file server.
- The end user sees the file server as just another hard disk. Because only the data storage input/output (I/O) is handled by the file server's computer, the MPSD offers limited capabilities for distributed processing.
- The end user must make a direct reference to the file server in order to access remote data. All record- and file-locking activities are done at the end-user location.
- All data selection, search, and update functions take place at the workstation, thus requiring that entire files travel through the network for processing at the workstation. Such a requirement increases network traffic, slows response time, and increases communication costs.

**FIGURE 12.7** Multiple-site processing, single-site data




The inefficiency of the last condition can be illustrated easily. For example, suppose that the file server computer stores a CUSTOMER table containing 10,000 data rows, 50 of which have balances greater than $1,000. Suppose that site A issues the following SQL query:

```
SELECT      *
FROM        CUSTOMER
WHERE       CUS_BALANCE > 1000;
```

All 10,000 CUSTOMER rows must travel through the network to be evaluated at site A. A variation of the multiple-site processing, single-site data approach is known as client/server architecture. **Client/server architecture** is similar to that of the network file server *except that all database processing is done at the server site, thus reducing network traffic*. Although both the network file server and the client/server systems perform multiple-site processing, the latter's processing is distributed. Note that the network file server approach requires the database to be located at a single site. In contrast, the client/server architecture is capable of supporting data at multiple sites.

**ONLINE CONTENT**

**Appendix F, Client/Server Systems,** is located in the Premium Website for this book.

### 12.6.3 MULTIPLE-SITE PROCESSING, MULTIPLE-SITE DATA (MPMD)

The **multiple-site processing, multiple-site data** (**MPMD**) scenario describes a fully distributed DBMS with support for multiple data processors and transaction processors at multiple sites. Depending on the level of support for various types of centralized DBMSs, DDBMSs are classified as either homogeneous or heterogeneous.

**Homogeneous DDBMSs** integrate only one type of centralized DBMS over a network. Thus, the same DBMS will be running on different server platforms (single processor server, multiprocessor server, server farms, or server blades). In contrast, **heterogeneous DDBMSs** integrate different types of centralized DBMSs over a network. Table 12.3 lists several systems that could be integrated within a single heterogeneous DDBMS over a network. A **fully heterogeneous DDBMS** will support different DBMSs that may even support different data models (relational, hierarchical, or network) running under different computer systems, such as mainframes and PCs.

TABLE 12.3 Heterogeneous Distributed Database scenario

| PLATFORM | DBMS | OPERATING SYSTEM | NETWORK COMMUNICATIONS PROTOCOL |
|---|---|---|---|
| **IBM 3090** | DB2 | MVS | APPC LU 6.2 |
| **DEC/VAX** | VAX rdb | OpenVMS | DECnet |
| **IBM AS/400** | SQL/400 | OS/400 | 3270 |
| **RISC Computer** | Informix | UNIX | TCP/IP |
| **Pentium CPU** | Oracle | Windows Server 2008 | TCP/IP |

Some DDBMS implementations support several platforms, operating systems, and networks and allow remote data access to another DBMS. However, such DDBMSs are still subject to certain restrictions. For example:

- Remote access is provided on a read-only basis and does not support write privileges.
- Restrictions are placed on the number of remote tables that may be accessed in a single transaction.
- Restrictions are placed on the number of distinct databases that may be accessed.
- Restrictions are placed on the database model that may be accessed. Thus, access may be provided to relational databases but not to network or hierarchical databases.

The preceding list of restrictions is by no means exhaustive. The DDBMS technology continues to change rapidly, and new features are added frequently. Managing data at multiple sites leads to a number of issues that must be addressed and understood. The next section will examine several key features of distributed database management systems.

## 12.7 DISTRIBUTED DATABASE TRANSPARENCY FEATURES

A distributed database system requires functional characteristics that can be grouped and described as transparency features. DDBMS transparency features have the common property of allowing the end user to feel like the database's only user. In other words, the user believes that (s)he is working with a centralized DBMS; all complexities of a distributed database are hidden, or transparent, to the user.

The DDBMS transparency features are:

- **Distribution transparency**, which allows a distributed database to be treated as a single logical database. If a DDBMS exhibits distribution transparency, the user does not need to know:
  - That the data are partitioned—meaning the table's rows and columns are split vertically or horizontally and stored among multiple sites.
  - That the data can be replicated at several sites.
  - The data location.
- **Transaction transparency**, which allows a transaction to update data at more than one network site. Transaction transparency ensures that the transaction will be either entirely completed or aborted, thus maintaining database integrity.
- **Failure transparency**, which ensures that the system will continue to operate in the event of a node failure. Functions that were lost because of the failure will be picked up by another network node.
- **Performance transparency**, which allows the system to perform as if it were a centralized DBMS. The system will not suffer any performance degradation due to its use on a network or due to the network's platform differences. Performance transparency also ensures that the system will find the most cost-effective path to access remote data.

- **Heterogeneity transparency**, which allows the integration of several different local DBMSs (relational, network, and hierarchical) under a common, or global, schema. The DDBMS is responsible for translating the data requests from the global schema to the local DBMS schema.

Distribution, transaction, and performance transparency will be examined in greater detail in the next few sections.

## 12.8 DISTRIBUTION TRANSPARENCY

Distribution transparency allows a physically dispersed database to be managed as though it were a centralized database. The level of transparency supported by the DDBMS varies from system to system. Three levels of distribution transparency are recognized:

- **Fragmentation transparency** is the highest level of transparency. The end user or programmer does not need to know that a database is partitioned. Therefore, neither fragment names nor fragment locations are specified prior to data access.
- **Location transparency** exists when the end user or programmer must specify the database fragment names but does not need to specify where those fragments are located.
- **Local mapping transparency** exists when the end user or programmer must specify both the fragment names and their locations.

Transparency features are summarized in Table 12.4.

**TABLE 12.4** A Summary of Transparency Features

| IF THE SQL STATEMENT REQUIRES: | | | |
|---|---|---|---|
| FRAGMENT NAME? | LOCATION NAME? | THEN THE DBMS SUPPORTS | LEVEL OF DISTRIBUTON TRANSPARENCY |
| Yes | Yes | Local mapping | Low |
| Yes | No | Location transparency | Medium |
| No | No | Fragmentation transparency | High |

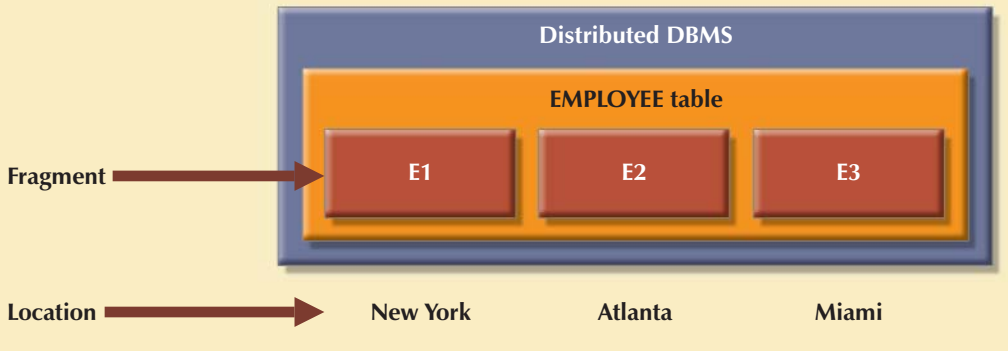
As you examine Table 12.4, you might ask why there is no reference to a situation in which the fragment name is "No" and the location name is "Yes." The reason for not including that scenario is simple: you cannot have a location name that fails to reference an existing fragment. (If you don't need to specify a fragment name, its location is clearly irrelevant.)

To illustrate the use of various transparency levels, suppose you have an EMPLOYEE table containing the attributes EMP_NAME, EMP_DOB, EMP_ADDRESS, EMP_DEPARTMENT, and EMP_SALARY. The EMPLOYEE data are distributed over three different locations: New York, Atlanta, and Miami. The table is divided by location; that is, New York employee data are stored in fragment E1, Atlanta employee data are stored in fragment E2, and Miami employee data are stored in fragment E3. (See Figure 12.8.)

Now suppose that the end user wants to list all employees with a date of birth prior to January 1, 1960. To focus on the transparency issues, also suppose that the EMPLOYEE table is fragmented and each fragment is unique. The **unique fragment** condition indicates that each row is unique, regardless of the fragment in which it is located. Finally, assume that no portion of the database is replicated at any other site on the network.

Depending on the level of distribution transparency support, you may examine three query cases.

FIGURE 12.8 Fragment locations




## Case 1: The Database Supports Fragmentation Transparency

The query conforms to a nondistributed database query format; that is, it does not specify fragment names or locations. The query reads:

```
SELECT      *
FROM        EMPLOYEE
WHERE       EMP_DOB < '01-JAN-196';
```

## Case 2: The Database Supports Location Transparency

Fragment names must be specified in the query, but the fragment's location is not specified. The query reads:

```
SELECT      *
FROM        E1
WHERE       EMP_DOB < '01-JAN-1960';
UNION
SELECT      *
FROM        E2
WHERE       EMP_DOB < '01-JAN-1960';
UNION
SELECT      *
FROM        E 3
WHERE       EMP_DOB < '01-JAN-1960';
```

## Case 3: The Database Supports Local Mapping Transparency

Both the fragment name and its location must be specified in the query. Using pseudo-SQL:

```
SELECT *
FROM El NODE NY
WHERE EMP_DOB < '01-JAN-1960';
UNION
SELECT *
FROM E2 NODE ATL
WHERE EMP_DOB < '01-JAN-1960';
UNION
SELECT *
```

```
FROM E3 NODE MIA
WHERE EMP_DOB < '01-JAN-1960';
```

**NOTE**

NODE indicates the location of the database fragment. NODE is used for illustration purposes and is not part of the standard SQL syntax.

As you examine the preceding query formats, you can see how distribution transparency affects the way end users and programmers interact with the database.

Distribution transparency is supported by a **distributed data dictionary** (**DDD**), or a **distributed data catalog** (**DDC**). The DDC contains the description of the entire database as seen by the database administrator. The database description, known as the **distributed global schema**, is the common database schema used by local TPs to translate user requests into subqueries (remote requests) that will be processed by different DPs. The DDC is itself distributed, and it is replicated at the network nodes. Therefore, the DDC must maintain consistency through updating at all sites.

Keep in mind that some of the current DDBMS implementations impose limitations on the level of transparency support. For instance, you might be able to distribute a database, but not a table, across multiple sites. Such a condition indicates that the DDBMS supports location transparency but not fragmentation transparency.

## 12.9 TRANSACTION TRANSPARENCY

Transaction transparency is a DDBMS property that ensures that database transactions will maintain the distributed database's integrity and consistency. Remember that a DDBMS database transaction can update data stored in many different computers connected in a network. Transaction transparency ensures that the transaction will be completed only when all database sites involved in the transaction complete their part of the transaction.

Distributed database systems require complex mechanisms to manage transactions and to ensure the database's consistency and integrity. To understand how the transactions are managed, you should know the basic concepts governing remote requests, remote transactions, distributed transactions, and distributed requests.

### 12.9.1 Distributed Requests and Distributed Transactions[1]

Whether or not a transaction is distributed, it is formed by one or more database requests. The basic difference between a nondistributed transaction and a distributed transaction is that the latter can update or request data from several different remote sites on a network. To better illustrate the distributed transaction concepts, let's begin by establishing the difference between remote and distributed transactions, using the BEGIN WORK and COMMIT WORK transaction format. Assume the existence of location transparency to avoid having to specify the data location.

A **remote request**, illustrated in Figure 12.9, lets a single SQL statement access the data that are to be processed by a single remote database processor. In other words, the SQL statement (or request) can reference data at only one remote site.

Similarly, a **remote transaction**, composed of several requests, accesses data at a single remote site. A remote transaction is illustrated in Figure 12.10.

[1] The details of distributed requests and transactions were originally described in David McGoveran and Colin White, "Clarifying Client/Server," *DBMS* 3(12), November 1990, pp. 78–89.

As you examine Figure 12.10, Note the following remote transaction features:

**FIGURE 12.9** A remote request

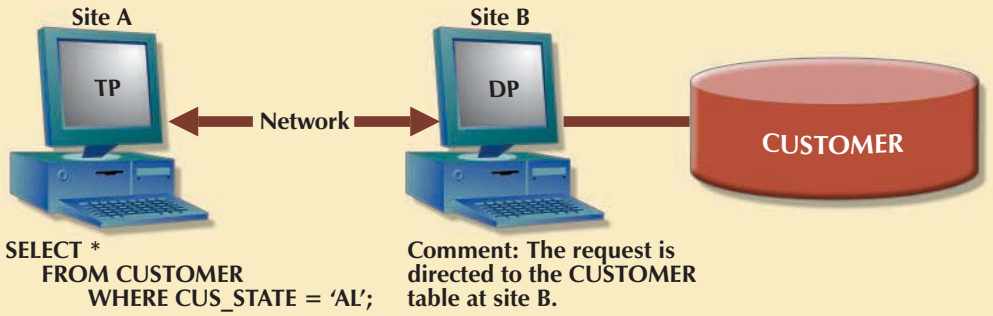
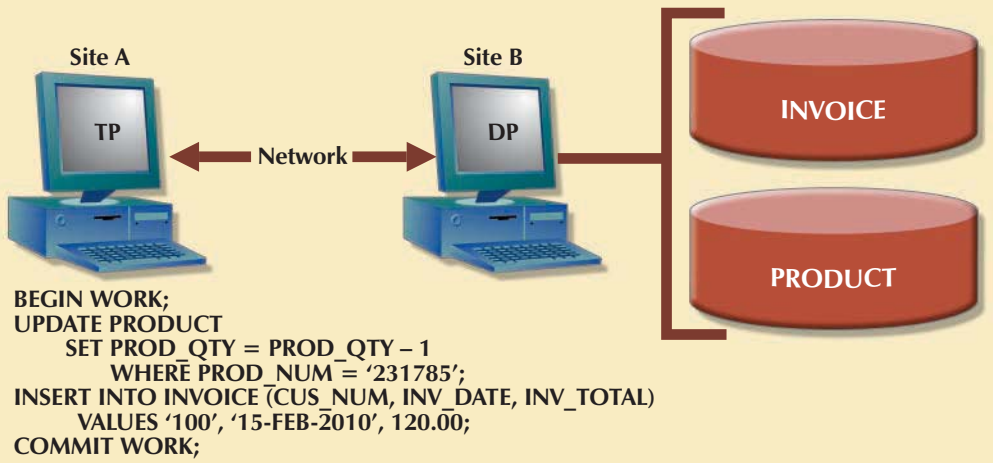


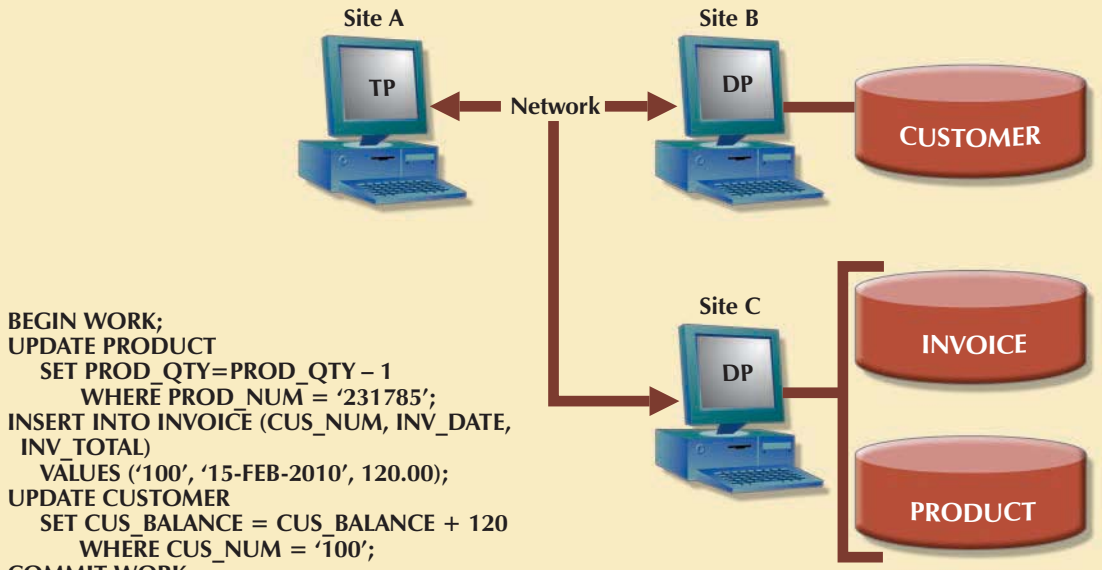

**FIGURE 12.10** A remote transaction




- The transaction updates the PRODUCT and INVOICE tables (located at site B).
- The remote transaction is sent to and executed at the remote site B.
- The transaction can reference only one remote DP.
- Each SQL statement (or request) can reference only one (the same) remote DP at a time, and the entire transaction can reference and be executed at only one remote DP.

A **distributed transaction** allows a transaction to reference several different local or remote DP sites. Although each single request can reference only one local or remote DP site, the transaction as a whole can reference multiple DP sites because each request can reference a different site. The distributed transaction process is illustrated in Figure 12.11.

**FIGURE 12.11** A distributed transaction




Note the following features in Figure 12.11:

- The transaction references two remote sites (B and C).
- The first two requests (UPDATE PRODUCT and INSERT INTO INVOICE) are processed by the DP at the remote site C, and the last request (UPDATE CUSTOMER) is processed by the DP at the remote site B.
- Each request can access only one remote site at a time.

The third characteristic may create problems. For example, suppose the table PRODUCT is divided into two fragments, PRODl and PROD2, located at sites B and C, respectively. Given that scenario, the preceding distributed transaction cannot be executed because the request:

```
SELECT      *
FROM        PRODUCT
WHERE       PROD_NUM = '231785';
```

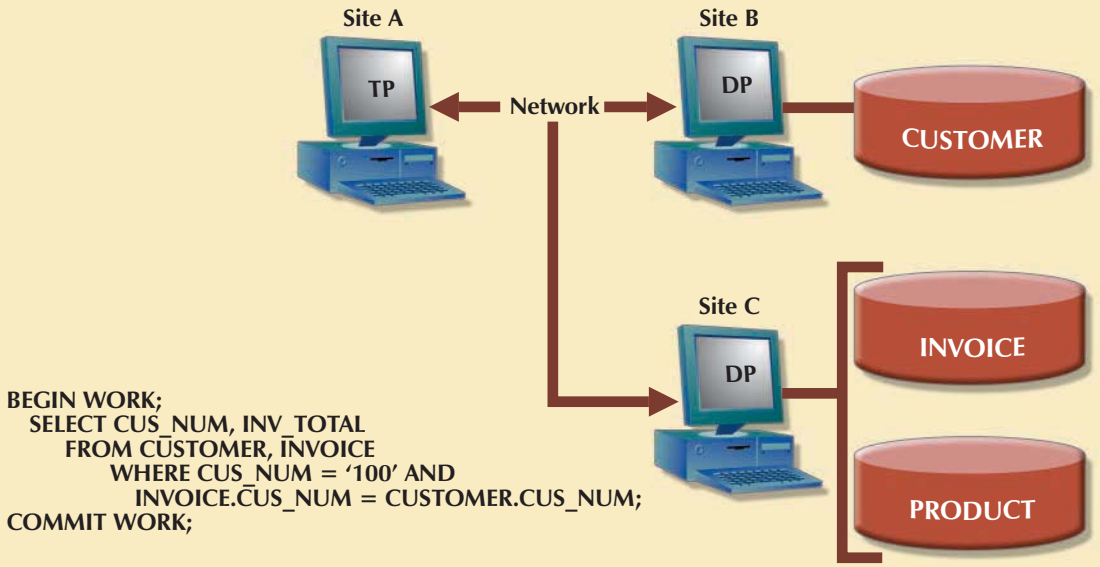
cannot access data from more than one remote site. Therefore, the DBMS must be able to support a distributed request.

A **distributed request** lets a single SQL statement reference data located at several different local or remote DP sites. Because each request (SQL statement) can access data from more than one local or remote DP site, a transaction can access several sites. The ability to execute a distributed request provides fully distributed database processing capabilities because of the ability to:

- Partition a database table into several fragments.
- Reference one or more of those fragments with only one request. In other words, there is fragmentation transparency.

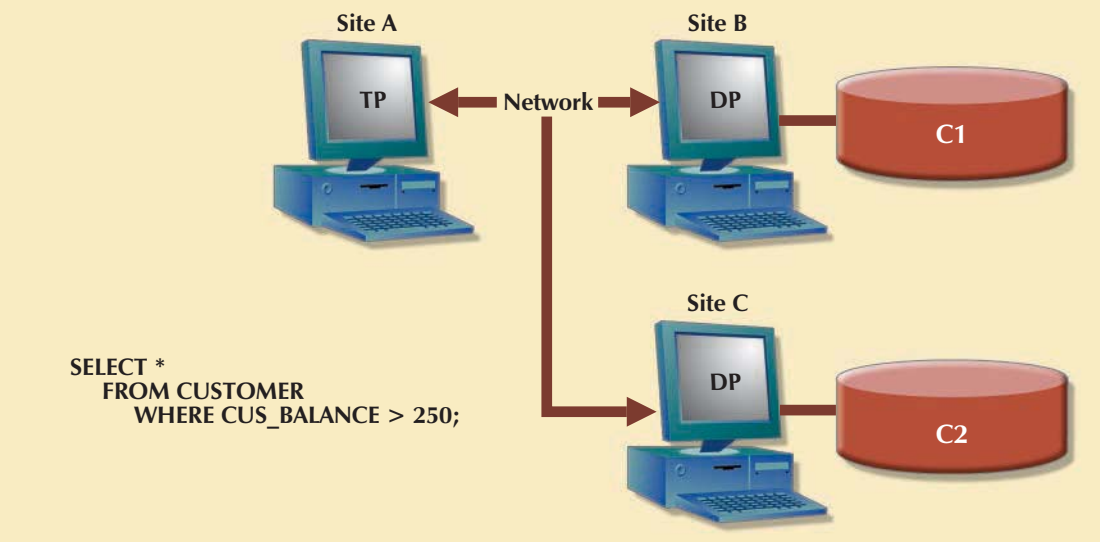
The location and partition of the data should be transparent to the end user. Figure 12.12 illustrates a distributed request. As you examine Figure 12.12, Note that the transaction uses a single SELECT statement to reference two tables, CUSTOMER and INVOICE. The two tables are located at two different sites, B and C.

**FIGURE 12.12** A distributed request




The distributed request feature also allows a single request to reference a physically partitioned table. For example, suppose that a CUSTOMER table is divided into two fragments, C1 and C2, located at sites B and C, respectively. Further suppose that the end user wants to obtain a list of all customers whose balances exceed $250. The request is illustrated in Figure 12.13. Full fragmentation transparency support is provided only by a DDBMS that supports distributed requests.

**FIGURE 12.13** Another distributed request



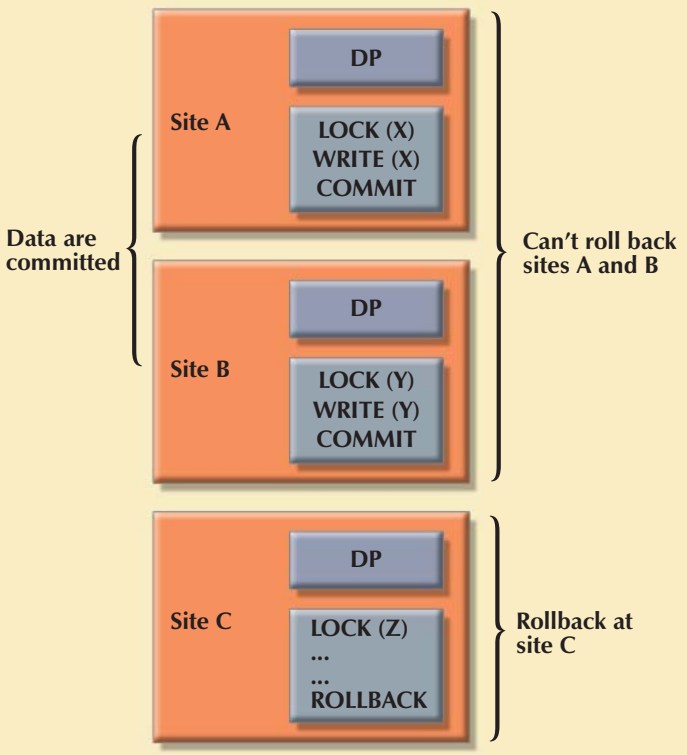
Understanding the different types of database requests in distributed database systems helps you address the transaction transparency issue more effectively. Transaction transparency ensures that distributed transactions are treated as centralized transactions, ensuring the serializability of transactions. (Review Chapter 10, Transaction Management and Concurrency Control, if necessary.) That is, the execution of concurrent transactions, whether or not they are distributed, will take the database from one consistent state to another.

### 12.9.2 Distributed Concurrency Control

Concurrency control becomes especially important in the distributed database environment because multisite, multiple-process operations are more likely to create data inconsistencies and deadlocked transactions than single-site systems are. For example, the TP component of a DDBMS must ensure that all parts of the transaction are completed at all sites before a final COMMIT is issued to record the transaction.

Suppose that each transaction operation was committed by each local DP, but one of the DPs could not commit the transaction's results. Such a scenario would yield the problems illustrated in Figure 12.14: the transaction(s) would yield an inconsistent database, with its inevitable integrity problems, because committed data cannot be uncommitted! The solution for the problem illustrated in Figure 12.14 is a *two-phase commit protocol*, which you will explore next.

**FIGURE 12.14** The effect of a premature COMMIT




### 12.9.3 Two-Phase Commit Protocol

Centralized databases require only one DP. All database operations take place at only one site, and the consequences of database operations are immediately known to the DBMS. In contrast, distributed databases make it possible for a transaction to access data at several sites. A final COMMIT must not be issued until all sites have committed their parts of the transaction. The **two-phase commit protocol** guarantees that if a portion of a transaction operation cannot

be committed; all changes made at the other sites participating in the transaction will be undone to maintain a consistent database state.

Each DP maintains its own transaction log. The two-phase commit protocol requires that the transaction entry log for each DP be written before the database fragment is actually updated. (See Chapter 10.) Therefore, the two-phase commit protocol requires a DO-UNDO-REDO protocol and a write-ahead protocol.

The **DO-UNDO-REDO protocol** is used by the DP to roll back and/or roll forward transactions with the help of the system's transaction log entries. The DO-UNDO-REDO protocol defines three types of operations:

- DO performs the operation and records the "before" and "after" values in the transaction log.
- UNDO reverses an operation, using the log entries written by the DO portion of the sequence.
- REDO redoes an operation, using the log entries written by the DO portion of the sequence.

To ensure that the DO, UNDO, and REDO operations can survive a system crash while they are being executed, a write-ahead protocol is used. The **write-ahead protocol** forces the log entry to be written to permanent storage before the actual operation takes place.

The two-phase commit protocol defines the operations between two types of nodes: the **coordinator** and one or more **subordinates**, or *cohorts*. The participating nodes agree on a coordinator. Generally, the coordinator role is assigned to the node that initiates the transaction. However, different systems implement various, more sophisticated election methods. The protocol is implemented in two phases:

### Phase 1: Preparation

The coordinator sends a PREPARE TO COMMIT message to all subordinates.

1. The subordinates receive the message; write the transaction log, using the write-ahead protocol; and send an acknowledgment (YES/PREPARED TO COMMIT or NO/NOT PREPARED) message to the coordinator.
2. The coordinator makes sure that all nodes are ready to commit, or it aborts the action.

If all nodes are PREPARED TO COMMIT, the transaction goes to Phase 2. If one or more nodes reply NO or NOT PREPARED, the coordinator broadcasts an ABORT message to all subordinates.

### Phase 2: The Final COMMIT

1. The coordinator broadcasts a COMMIT message to all subordinates and waits for the replies.
2. Each subordinate receives the COMMIT message, and then updates the database using the DO protocol.
3. The subordinates reply with a COMMITTED or NOT COMMITTED message to the coordinator.

If one or more subordinates did not commit, the coordinator sends an ABORT message, thereby forcing them to UNDO all changes.

The objective of the two-phase commit is to ensure that each node commits its part of the transaction; otherwise, the transaction is aborted. If one of the nodes fails to commit, the information necessary to recover the database is in the transaction log, and the database can be recovered with the DO-UNDO-REDO protocol. (Remember that the log information was updated using the write-ahead protocol.)

## 12.10 PERFORMANCE TRANSPARENCY AND QUERY OPTIMIZATION

One of the most important functions of a database is its ability to make data available. Because all data reside at a single site in a centralized database, the DBMS must evaluate every data request and find the most efficient way to access the local data. In contrast, the DDBMS makes it possible to partition a database into several fragments, thereby rendering

the query translation more complicated, because the DDBMS must decide which fragment of the database to access. In addition, the data may also be replicated at several different sites. The data replication makes the access problem even more complex, because the database must decide which copy of the data to access. The DDBMS uses query optimization techniques to deal with such problems and to ensure acceptable database performance.

The objective of a query optimization routine is to minimize the total cost associated with the execution of a request. The costs associated with a request are a function of the:

- Access time (I/O) cost involved in accessing the physical data stored on disk.
- Communication cost associated with the transmission of data among nodes in distributed database systems.
- CPU time cost associated with the processing overhead of managing distributed transactions.

Although costs are often classified as either communication or processing costs, it is difficult to separate the two. Not all query optimization algorithms use the same parameters, and all algorithms do not assign the same weight to each parameter. For example, some algorithms minimize total time; others minimize the communication time, and still others do not factor in the CPU time, considering it insignificant relative to other cost sources.

**NOTE**

**Chapter 11, Database Performance Tuning and Query Optimization**, provides additional details about query optimization.

To evaluate query optimization, keep in mind that the TP must receive data from the DP, synchronize it, assemble the answer, and present it to the end user or an application. Although that process is standard, you should consider that a particular query may be executed at any one of several different sites. The response time associated with remote sites cannot be easily predetermined because some nodes are able to finish their part of the query in less time than others.

One of the most important characteristics of query optimization in distributed database systems is that it must provide distribution transparency as well as *replica* transparency. (Distribution transparency was explained earlier in this chapter.) **Replica transparency** refers to the DDBMS's ability to hide the existence of multiple copies of data from the user.

Most of the algorithms proposed for query optimization are based on two principles:

- The selection of the optimum execution order.
- The selection of sites to be accessed to minimize communication costs.

Within those two principles, a query optimization algorithm can be evaluated on the basis of its *operation mode* or the *timing of its optimization*.

Operation modes can be classified as manual or automatic. **Automatic query optimization** means that the DDBMS finds the most cost-effective access path without user intervention. **Manual query optimization** requires that the optimization be selected and scheduled by the end user or programmer. Automatic query optimization is clearly more desirable from the end user's point of view, but the cost of such convenience is the increased overhead that it imposes on the DDBMS.

Query optimization algorithms can also be classified according to when the optimization is done. Within this timing classification, query optimization algorithms can be classified as static or dynamic.

- **Static query optimization** takes place at compilation time. In other words, the best optimization strategy is selected when the query is compiled by the DBMS. This approach is common when SQL statements are embedded in procedural programming languages such as C# or Visual Basic .NET. When the program is submitted to the DBMS for compilation, it creates the plan necessary to access the database. When the program is executed, the DBMS uses that plan to access the database.

- **Dynamic query optimization** takes place at execution time. Database access strategy is defined when the program is executed. Therefore, access strategy is dynamically determined by the DBMS at run time, using the most up-to-date information about the database. Although dynamic query optimization is efficient, its cost is measured by run-time processing overhead. The best strategy is determined every time the query is executed; this could happen several times in the same program.

Finally, query optimization techniques can be classified according to the type of information that is used to optimize the query. For example, queries may be based on statistically based or rule-based algorithms.

- A **statistically based query optimization algorithm** uses statistical information about the database. The statistics provide information about database characteristics such as size, number of records, average access time, number of requests serviced, and number of users with access rights. These statistics are then used by the DBMS to determine the best access strategy.
- The statistical information is managed by the DDBMS and is generated in one of two different modes: dynamic or manual. In the **dynamic statistical generation mode**, the DDBMS automatically evaluates and updates the statistics after each access. In the **manual statistical generation mode**, the statistics must be updated periodically through a user-selected utility such as IBM's RUNSTAT command used by DB2 DBMSs.
- A **rule-based query optimization algorithm** is based on a set of user-defined rules to determine the best query access strategy. The rules are entered by the end user or database administrator, and they are typically very general in nature.

## 12.11 DISTRIBUTED DATABASE DESIGN

Whether the database is centralized or distributed, the design principles and concepts described in Chapter 3, The Relational Database Model; Chapter 4, Entity Relationship (ER) Modeling; and Chapter 6, Normalization of Database Tables, are still applicable. However, the design of a distributed database introduces three new issues:

- How to partition the database into fragments.
- Which fragments to replicate.
- Where to locate those fragments and replicas.

Data fragmentation and data replication deal with the first two issues, and data allocation deals with the third issue.

### 12.11.1 Data Fragmentation

**Data fragmentation** allows you to break a single object into two or more segments, or fragments. The object might be a user's database, a system database, or a table. Each fragment can be stored at any site over a computer network. Information about data fragmentation is stored in the distributed data catalog (DDC), from which it is accessed by the TP to process user requests.

Data fragmentation strategies, as discussed here, are based at the table level and consist of dividing a table into logical fragments. You will explore three types of data fragmentation strategies: horizontal, vertical, and mixed. (Keep in mind that a fragmented table can always be re-created from its fragmented parts by a combination of unions and joins.)

- **Horizontal fragmentation** refers to the division of a relation into subsets (fragments) of tuples (rows). Each fragment is stored at a different node, and each fragment has unique rows. However, the unique rows all have the same attributes (columns). In short, each fragment represents the equivalent of a SELECT statement, with the WHERE clause on a single attribute.
- **Vertical fragmentation** refers to the division of a relation into attribute (column) subsets. Each subset (fragment) is stored at a different node, and each fragment has unique columns—with the exception of the key column, which is common to all fragments. This is the equivalent of the PROJECT statement in SQL.
- **Mixed fragmentation** refers to a combination of horizontal and vertical strategies. In other words, a table may be divided into several horizontal subsets (rows), each one having a subset of the attributes (columns).