CHAPTER

6

Introduction to SQL

Visit www.pearsonhighered.com/hoffer to view the accompanying video for this chapter.

Learning Objectives

After studying this chapter, you should be able to:

- Concisely define each of the following key terms: **relational DBMS (RDBMS), catalog, schema, data definition language (DDL), data manipulation language (DML), data control language (DCL), referential integrity, scalar aggregate, vector aggregate, base table, virtual table, dynamic view,** and **materialized view.**
- Interpret the history and role of SQL in database development.
- Define a database using the SQL data definition language.
- Write single-table queries using SQL commands.
- Establish referential integrity using SQL.
- Discuss the SQL:1999 and SQL:200n standards.

INTRODUCTION

Pronounced "S-Q-L" by some and "sequel" by others, SQL has become the de facto standard language for creating and querying relational databases. (Can the next standard be the sequel to SQL?) The primary purpose of this chapter is to introduce SQL, the most common language for relational systems. It has been accepted as a U.S. standard by the American National Standards Institute (ANSI) and is a Federal Information Processing Standard (FIPS). It is also an international standard recognized by the International Organization for Standardization (ISO). ANSI has accredited the International Committee for Information Technology Standards (INCITS) as a standards development organization; INCITS is working on the next version of the SQL standard to be released.

The SQL standard is like afternoon weather in Florida (and maybe where you live, too)—wait a little while, and it will change. The ANSI SQL standards were first published in 1986 and updated in 1989, 1992 (SQL-92), 1999 (SQL:1999), 2003 (SQL:2003), 2006 (SQL:2006), and 2008 (SQL:2008). SQL:2008 was in final draft form at the time of writing this edition. (See **http://en.wikipedia.org/wiki/SQL** for a summary of this history.) The standard is now generally referred to as SQL:200n (they will need SQL:20nn any day now!).

SQL-92 was a major revision and was structured into three levels: Entry, Intermediate, and Full. SQL:1999 established core-level conformance, which must be met before any other level of conformance can be achieved; core-level conformance requirements are unchanged in SQL:200n. In addition to fixes and enhancements of SQL:1999, SQL:2003 introduced a new set of SQL/XML standards,

three new data types, various new built-in functions, and improved methods for generating values automatically. SQL:2006 refined these additions and made them more compatible with XQuery, the XML query language published by the World Wide Web Consortium (W3C). At the time of this writing, most database management systems claim SQL:1992 compliance and partial compliance with SQL:1999 and SQL:200n.

Except where noted as a particular vendor's syntax, the examples in this chapter conform to the SQL standard. Concerns have been expressed about SQL:1999 and SQL:2003/SQL:200n being true standards because conformance with the standard is no longer certified by the U.S. Department of Commerce's National Institute of Standards and Technology (NIST) (Gorman, 2001). "Standard SQL" may be considered an oxymoron (like safe investment or easy payments)! Vendors' interpretations of the SQL standard differ from each other, and vendors extend their products' capabilities with proprietary features beyond the stated standard. This makes it difficult to port SQL from one vendor's product to another. One must become familiar with the particular version of SQL being used and not expect that SQL code will transfer exactly as written to another vendor's version. Table 6-1 demonstrates differences in handling date and time values to illustrate discrepancies one encounters across SQL vendors (IBM DB2, Microsoft SQL Server, MySQL [an open source DBMS], and Oracle).

SQL has been implemented in both mainframe and personal computer systems, so this chapter is relevant to both computing environments. Although many of the PC-database packages use a query-by-example (QBE) interface, they also include SQL coding as an option. QBE interfaces use graphic presentations and translate the QBE actions into SQL code before query execution occurs. In Microsoft Access, for example, it is possible to switch back and forth between the two interfaces; a query that has been built using a QBE interface can be viewed in SQL by clicking a button. This feature may aid you in learning SQL syntax. In client/server architectures, SQL commands are executed on the server, and the results are returned to the client workstation.

The first commercial DBMS that supported SQL was Oracle in 1979. Oracle is now available in mainframe, client/server, and PC-based platforms for many operating systems, including various UNIX, Linux, and Microsoft Windows operating systems. IBM's DB2, Informix, and Microsoft SQL Server are available for this range of operating systems also. See Eisenberg et al. (2004) for an overview of SQL:2003.

TABLE 6-1 Handling Date and Time Values (Arvin, 2005,based on content currently and previously available at http://troelsarvin.blogspot.com/)

TIMESTAMP data type: A core feature, the standard requires that this data type store year, month, day, hour, minute, and second (with fractional seconds; default is six digits).

TIMESTAMP WITH TIME ZONE data type: Extension to TIMESTAMP also stores the time zone.

		Implementation:
Product	**Follows Standard?**	**Comments**
DB2	TIMESTAMP only	Includes validity check and will not accept an entry such as 2010–02–29 00:05:00.
MS-SQL	No	DATETIME stores date and time, with only three digits for fractional seconds; DATETIME2 has a larger date range and greater fractional precision. Validity check similar to DB2's is included.
MySQL	No	TIMESTAMP updates to current date and time when other data in the row are updated and displays the value for the time zone of the user. DATETIME similar to MS-SQL, but validity checking is less accurate and may result in values of zero being stored.
Oracle	TIMESTAMP and TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE not allowed as part of a unique key. Includes validity check on dates.

ORIGINS OF THE SQL STANDARD

The concepts of relational database technology were first articulated in 1970, in E. F. Codd's classic paper "A Relational Model of Data for Large Shared Data Banks." Workers at the IBM Research Laboratory in San Jose, California, undertook development of System R, a project whose purpose was to demonstrate the feasibility of implementing the relational model in a database management system. They used a language called Sequel, also developed at the San Jose IBM Research Laboratory. Sequel was renamed SQL during the project, which took place from 1974 to 1979. The knowledge gained was applied in the development of SQL/DS, the first relational database management system available commercially (from IBM). SQL/DS was first available in 1981, running on the DOS/VSE operating system. A VM version followed in 1982, and the MVS version, DB2, was announced in 1983.

When System R was well received at the user sites where it was installed, other vendors began developing relational products that used SQL. One product, Oracle, from Relational Software, was actually on the market before SQL/DS (1979). Other products included INGRES from Relational Technology (1981), IDM from Britton-Lee (1982), DG/SQL from Data General Corporation (1984), and Sybase from Sybase, Inc. (1986). To provide some directions for the development of relational DBMSs, ANSI and the ISO approved a standard for the SQL relational query language (functions and syntax) that was originally proposed by the X3H2 Technical Committee on Database (Technical Committee X3H2—Database, 1986; ISO, 1987), often referred to as SQL/86. For a more detailed history of the SQL standard, see the documents available at **www.wiscorp.com**.

The following were the original purposes of the SQL standard:

1. To specify the syntax and semantics of SQL data definition and manipulation languages
2. To define the data structures and basic operations for designing, accessing, maintaining, controlling, and protecting an SQL database
3. To provide a vehicle for portability of database definition and application modules between conforming DBMSs
4. To specify both minimal (Level 1) and complete (Level 2) standards, which permit different degrees of adoption in products
5. To provide an initial standard, although incomplete, that will be enhanced later to include specifications for handling such topics as referential integrity, transaction management, user-defined functions, join operators beyond the equi-join, and national character sets

In terms of SQL, when is a standard not a standard? As explained earlier, most vendors provide unique, proprietary features and commands for their SQL database management system. So, what are the advantages and disadvantages of having an SQL standard, when there is such variations from vendor to vendor? The benefits of such a standardized relational language include the following (although these are not pure benefits because of vendor differences):

- ***Reduced training costs*** Training in an organization can concentrate on one language. A large labor pool of IS professionals trained in a common language reduces retraining for newly hired employees.
- ***Productivity*** IS professionals can learn SQL thoroughly and become proficient with it from continued use. An organization can afford to invest in tools to help IS professionals become more productive. And because they are familiar with the language in which programs are written, programmers can more quickly maintain existing programs.
- ***Application portability*** Applications can be moved from machine to machine when each machine uses SQL. Further, it is economical for the computer software industry to develop off-the-shelf application software when there is a standard language.
- ***Application longevity*** A standard language tends to remain so for a long time; hence there will be little pressure to rewrite old applications. Rather, applications

will simply be updated as the standard language is enhanced or new versions of DBMSs are introduced.

- ***Reduced dependence on a single vendor*** When a nonproprietary language is used, it is easier to use different vendors for the DBMS, training and educational services, application software, and consulting assistance; further, the market for such vendors will be more competitive, which may lower prices and improve service.
- ***Cross-system communication*** Different DBMSs and application programs can more easily communicate and cooperate in managing data and processing user programs.

On the other hand, a standard can stifle creativity and innovation; one standard is never enough to meet all needs, and an industry standard can be far from ideal because it may be the offspring of compromises among many parties. A standard may be difficult to change (because so many vendors have a vested interest in it), so fixing deficiencies may take considerable effort. Another disadvantage of standards that can be extended with proprietary features is that using special features added to SQL by a particular vendor, may result in the loss of some advantages, such as application portability.

The original SQL standard has been widely criticized, especially for its lack of referential integrity rules and certain relational operators. Date and Darwen (1997) express concern that SQL seems to have been designed without adhering to established principles of language design, and "as a result, the language is filled with numerous restrictions, ad hoc constructs, and annoying special rules" (p. 8). They feel that the standard is not explicit enough and that the problem of standard SQL implementations will continue to exist. Some of these limitations will be noticeable in this chapter.

Many products are available that support SQL, and they run on machines of all sizes, from small personal computers to large mainframes. The database market is maturing, and the rate of significant changes in products may slow, but they will continue to be SQL based. The number of relational database vendors with significant market share has continued to consolidate. According to Lai (2007), Oracle controlled over 44 percent of the overall database market in 2007, IBM a little over 21 percent, and Microsoft almost 19 percent. Sybase and Teradata also had significant—albeit much smaller—shares, and open source products, such as MySQL, PostgreSQL, and Ingres, combined for about 10 percent market share. MySQL, an open source version of SQL that runs on Linux, UNIX, Windows, and Mac OS X operating systems, has achieved considerable popularity. (Download MySQL for free from **www.mysql.com**.) The market position of MySQL may be changing; at the time of writing this edition, Oracle had just acquired MySQL as part of its purchase of Sun Microsystems. Opportunities still exist for smaller vendors to prosper through industry-specific systems or niche applications. Upcoming product releases may change the relative strengths of the database management systems by the time you read this book. But all of them will continue to use SQL, and they will follow, to a certain extent, the standard described here.

Because of its significant market share, we most often illustrate SQL in this text using Oracle 11g syntax. We illustrate using a specific relational DBMS not to promote or endorse Oracle but rather so we know that the code we use will work with some DBMS. In the vast majority of the cases, the code will, in fact, work with many relational DBMSs because it complies with standard ANSI SQL. In some cases, we include illustrations using several or other relational DBMSs when there are interesting differences; however, there are only a few such cases, because we are not trying to compare systems, and we want to be parsimonious.

THE SQL ENVIRONMENT

With today's relational DBMSs and application generators, the importance of SQL within the database architecture is not usually apparent to the application users. Many users who access database applications have no knowledge of SQL at all. For example, sites on the Web allow users to browse their catalogs (e.g., see **www.llbean.com**). The information about an item that is presented, such as size, color, description, and

availability, is stored in a database. The information has been retrieved using an SQL query, but the user has not issued an SQL command. Rather, the user has used a prewritten program (e.g., written in Java) with embedded SQL commands for database processing.

An SQL-based relational database application involves a user interface, a set of tables in the database, and a relational database management system (RDBMS) with an SQL capability. Within the RDBMS, SQL will be used to create the tables, translate user requests, maintain the data dictionary and system catalog, update and maintain the tables, establish security, and carry out backup and recovery procedures. A **relational DBMS (RDBMS)** is a data management system that implements a relational data model, one where data are stored in a collection of tables, and the data relationships are represented by common values, not links. This view of data was illustrated in Chapter 2 for the Pine Valley Furniture database system and will be used throughout this chapter's SQL query examples.

Relational DBMS (RDBMS)
A database management system that manages data as a collection of tables in which all data relationships are represented by common values in related tables.

Figure 6-1 is a simplified schematic of an SQL environment, consistent with SQL:200n standard. As depicted, an SQL environment includes an instance of an SQL database management system along with the databases accessible by that DBMS and the users and programs that may use that DBMS to access the databases. Each database is contained in a **catalog**, which describes any object that is a part of the database, regardless of which user created that object. Figure 6-1 shows two catalogs: DEV_C and PROD_C. Most companies keep at least two versions of any database they are using. The production version, PROD_C here, is the live version, which captures real business data and thus must be very tightly controlled and monitored. The development version, DEV_C here, is used when the database is being built and continues to serve as a development tool where enhancements and maintenance efforts can be thoroughly tested before being applied to the production database. Typically this database is not as tightly controlled or monitored, because it does not contain live business data. Each database will have a named schema(s) associated with a catalog. A **schema** is a collection of related objects, including but not limited to base tables and views, domains, constraints, character sets, triggers, and roles.

Catalog
A set of schemas that, when put together, constitute a description of a database.

If more than one user has created objects in a database, combining information about all users' schemas will yield information for the entire database. Each catalog must also contain an information schema, which contains descriptions of all schemas in the catalog, tables, views, attributes, privileges, constraints, and domains, along with

Schema
A structure that contains descriptions of objects created by a user, such as base tables, views, and constraints, as part of a database.

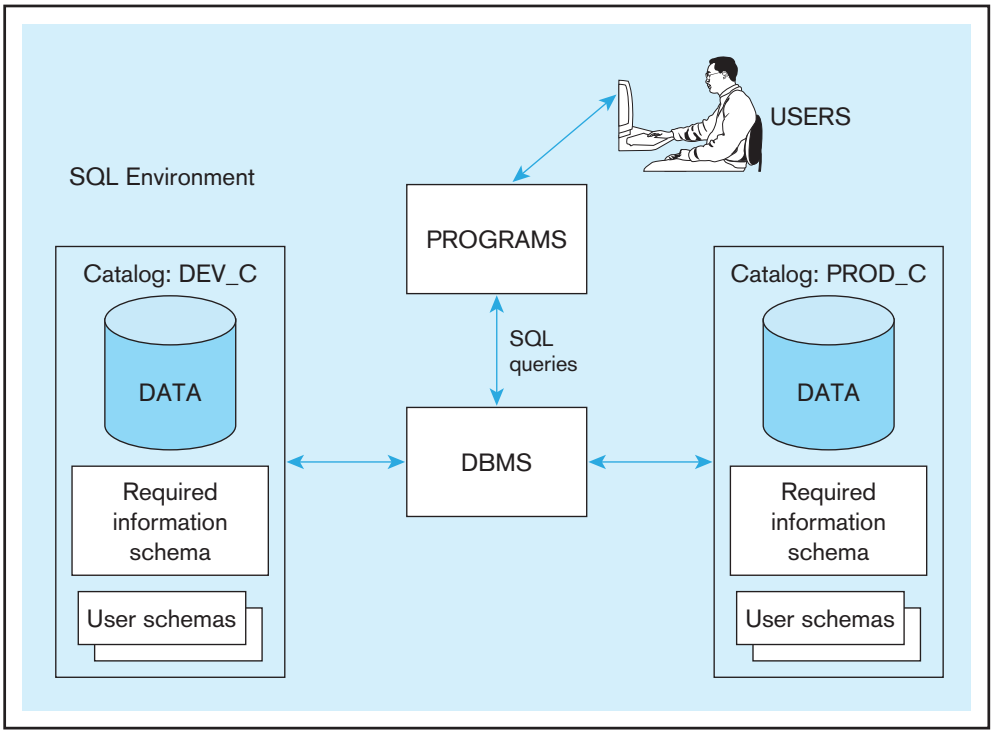


FIGURE 6-1 A simplified schematic of a typical SQL environment, as described by the SQL:2000n standards

other information relevant to the database. The information contained in the catalog is maintained by the DBMS as a result of the SQL commands issued by the users and can be rebuilt without conscious action by the user. It is part of the power of the SQL language that the issuance of syntactically simple SQL commands may result in complex data management activities being carried out by the DBMS software. Users can browse the catalog contents by using SQL SELECT statements.

Data definition language (DDL)
Commands used to define a database, including those for creating, altering, and dropping tables and establishing constraints.

SQL commands can be classified into three types. First, there are **data definition language (DDL)** commands. These commands are used to create, alter, and drop tables, views, and indexes, and they are covered first in this chapter. There may be other objects controlled by the DDL, depending on the DBMS. For example, many DBMSs support defining synonyms (abbreviations) for database objects or a field to hold a specified sequence of numbers (which can be helpful in assigning primary keys to rows in tables). In a production database, the ability to use DDL commands will generally be restricted to one or more database administrators in order to protect the database structure from unexpected and unapproved changes. In development or student databases, DDL privileges will be granted to more users.

Data manipulation language (DML)
Commands used to maintain and query a database, including those for updating, inserting, modifying, and querying data.

Next, there are **data manipulation language (DML)** commands. Many consider the DML commands to be the core commands of SQL. These commands are used for updating, inserting, modifying, and querying the data in the database. They may be issued interactively, so that a result is returned immediately following the execution of the statement, or they may be included within programs written in a procedural programming language, such as C, Java, PHP, or COBOL or with a GUI tool (e.g., SQL Assistant with Teradata or MySQL Query Browser). Embedding SQL commands may provide the programmer with more control over timing of report generation, interface appearance, error handling, and database security (see Chapter 8 on embedding SQL in Web-based programs). Most of this chapter is devoted to covering basic DML commands, in interactive format. The general syntax of the SQL SELECT command used in DML is shown in Figure 6-2.

Data control language (DCL)
Commands used to control a database, including those for administering privileges and committing (saving) data.

Finally, **data control language (DCL)** commands help a DBA control the database; they include commands to grant or revoke privileges to access the database or particular objects within the database and to store or remove transactions that would affect the database.

Each DBMS has a defined list of data types that it can handle. All contain numeric, string, and date/time-type variables. Some also contain graphic data types, spatial data types, or image data types, which greatly increase the flexibility of data manipulation. When a table is created, the data type for each attribute must be specified. Selection of a particular data type is affected by the data values that need to be stored and the expected uses of the data. A unit price will need to be stored in a numeric format because mathematical manipulations such as multiplying unit price by the number of units ordered are expected. A phone number may be stored as string data, especially if foreign phone numbers are going to be included in the data set. Even though a phone number contains only digits, no mathematical operations, such as adding or multiplying phone numbers, make sense with a phone number. And because character data will process more quickly, numeric data should be stored as character data if no arithmetic calculations are expected. Selecting a date field rather than a string field will allow the developer to take advantage of date/time interval calculation functions that cannot be applied to a character field. See Table 6-2 for a few examples of SQL data types. SQL:200n includes three new data types: BIGINT, MULTISET, and XML. Watch for

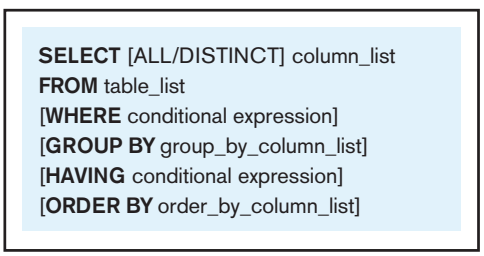


FIGURE 6-2 General syntax of the SELECT statement used in DML

TABLE 6-2 Sample SQL Data Types

String	CHARACTER (CHAR)	Stores string values containing any characters in a character set. CHAR is defined to be a fixed length.
	CHARACTER VARYING (VARCHAR or VARCHAR2)	Stores string values containing any characters in a character set but of definable variable length.
	BINARY LARGE OBJECT (BLOB)	Stores binary string values in hexadecimal format. BLOB is defined to be a variable length. (Oracle also has CLOB and NCLOB, as well as BFILE for storing unstructured data outside the database.)
Number	NUMERIC	Stores exact numbers with a defined precision and scale.
	INTEGER (INT)	Stores exact numbers with a predefined precision and scale of zero.
Temporal	TIMESTAMPTIMESTAMP WITH LOCAL TIME ZONE	Stores a moment an event occurs, using a definable fraction-of–a-second precision.Value adjusted to the user's session time zone (available in Oracle and MySQL)
Boolean	BOOLEAN	Stores truth values: TRUE, FALSE, or UNKNOWN.

these new data types to be added to RDBMSs that had not previously introduced them as an enhancement of the existing standard.

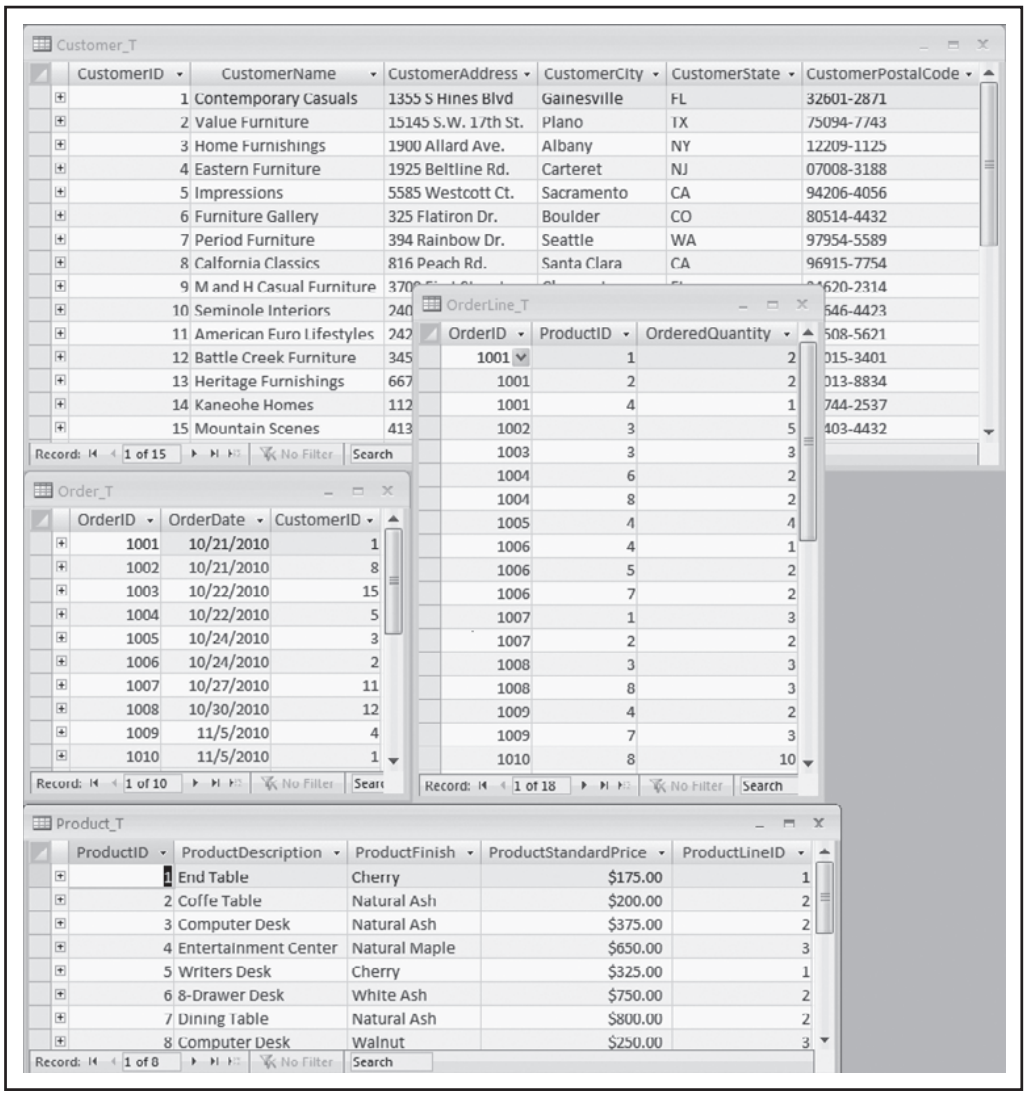
Given the wealth of graphic and image data types, it is necessary to consider business needs when deciding how to store data. For example, color may be stored as a descriptive character field, such as "sand drift" or "beige." But such descriptions will vary from vendor to vendor and do not contain the amount of information that could be contained in a spatial data type that includes exact red, green, and blue intensity values. Such data types are now available in universal servers, which handle data warehouses, and can be expected to appear in RDBMSs as well. In addition to the predefined data types included in Table 6-2, SQL:1999 and SQL:200n support constructed data types and user-defined types. There are many more predefined data types than those shown in Table 6-2. It will be necessary to familiarize yourself with the available data types for each RDBMS with which you work to achieve maximum advantage from its capabilities.

We are almost ready to illustrate sample SQL commands. The sample data that we will be using are shown in Figure 6-3 (which was captured in Microsoft Access). The data model corresponds to that shown in Figure 2-22. The PVFC database files are available for your use on this book's Web site; the files are available in several formats, for use with different DBMSs, and the database is also available on Teradata Student Network. Instructions for locating them are included inside the front cover of the book. There are two PVFC files. The one used here is named BookPVFC (also called Standard PVFC), and you can use it to work through the SQL queries demonstrated in Chapters 6 and 7. Another file, BigPVFC, contains more data and does not always correspond to Figure 2-22, nor does it always demonstrate good database design. Big PVFC is used for some of the exercises at the end of the chapter.

Each table name follows a naming standard that places an underscore and the letter T (for table) at the end of each table name, such as Order_T or Product_T. (Most DBMSs do not permit a space in the name of a table nor typically in the name of an attribute.) When looking at these tables, note the following:

1. Each order must have a valid customer ID included in the Order_T table.
2. Each item in an order line must have both a valid product ID and a valid order ID associated with it in the OrderLine_T table.
3. These four tables represent a simplified version of one of the most common sets of relations in business database systems—the customer order for products. SQL commands necessary to create the Customer_T table and the Order_T table were included in Chapter 2 and are expanded here.

FIGURE 6-3 Sample Pine Valley Furniture Company data



Customer_T

CustomerID	CustomerName	CustomerAddress	CustomerCity	CustomerState	CustomerPostalCode
1	Contemporary Casuals	1355 S Hines Blvd	Gainesville	FL	32601-2871
2	Value Furniture	15145 S.W. 17th St.	Plano	TX	75094-7743
3	Home Furnishings	1900 Allard Ave.	Albany	NY	12209-1125
4	Eastern Furniture	1925 Beltline Rd.	Carteret	NJ	07008-3188
5	Impressions	5585 Westcott Ct.	Sacramento	CA	94206-4056
6	Furniture Gallery	325 Flatiron Dr.	Boulder	CO	80514-4432
7	Period Furniture	394 Rainbow Dr.	Seattle	WA	97954-5589
8	California Classics	816 Peach Rd.	Santa Clara	CA	96915-7754
9	M and H Casual Furniture	3700			94620-2314
10	Seminole Interiors	240			546-4423
11	American Euro Lifestyles	242			508-5621
12	Battle Creek Furniture	345			015-3401
13	Heritage Furnishings	667			013-8834
14	Kaneohe Homes	112			744-2537
15	Mountain Scenes	413			403-4432

Record: 1 of 15 No Filter Search

OrderLine_T

OrderID	ProductID	OrderedQuantity
1001	1	2
1001	2	2
1001	4	1
1002	3	5
1003	3	3
1004	6	2
1004	8	2
1005	4	4
1006	4	1
1006	5	2
1006	7	2
1007	1	3
1007	2	2
1008	3	3
1008	8	3
1009	4	2
1009	7	3
1010	8	10

Record: 1 of 18 No Filter Search

Order_T

OrderID	OrderDate	CustomerID
1001	10/21/2010	1
1002	10/21/2010	8
1003	10/22/2010	15
1004	10/22/2010	5
1005	10/24/2010	3
1006	10/24/2010	2
1007	10/27/2010	11
1008	10/30/2010	12
1009	11/5/2010	4
1010	11/5/2010	1

Record: 1 of 10 No Filter Search

Product_T

ProductID	ProductDescription	ProductFinish	ProductStandardPrice	ProductLineID
1	End Table	Cherry	$175.00	1
2	Coffe Table	Natural Ash	$200.00	2
3	Computer Desk	Natural Ash	$375.00	2
4	Entertainment Center	Natural Maple	$650.00	3
5	Writers Desk	Cherry	$325.00	1
6	8-Drawer Desk	White Ash	$750.00	2
7	Dining Table	Natural Ash	$800.00	2
8	Computer Desk	Walnut	$250.00	3

Record: 1 of 8 No Filter Search

The remainder of the chapter will illustrate DDL, DML, and DCL commands. Figure 6-4 gives an overview of where the various types of commands are used throughout the database development process. We will use the following notation in the illustrative SQL commands:

1. All-capitalized words denote commands. Type them exactly as shown, though capitalization may not be required by the RDBMSs. Some RDBMSs will always show data names in output using all capital letters, even if they can be entered in mixed case. (This is the style of Oracle, which is what we follow except where noted.) Tables, columns, named constraints, and so forth are shown in mixed case. Remember that table names follow the "underscore T" convention. SQL commands do not have an "underscore" and so should be easy to distinguish from table and column names. Also, RDBMSs do not like embedded spaces in data names, so multiple-word data names from ERDs are entered with the words together, without spaces between them. A consequence is that, for example, a column named QtyOnHand will become QTYONHAND when it is displayed by many RDBMSs. (You can use the ALIAS clause in a SELECT to rename a column name to a more readable value for display.)
2. Lowercase and mixed-case words denote values that must be supplied by the user.
3. Brackets enclose optional syntax.
4. An ellipsis (. . .) indicates that the accompanying syntactic clause may be repeated as necessary.

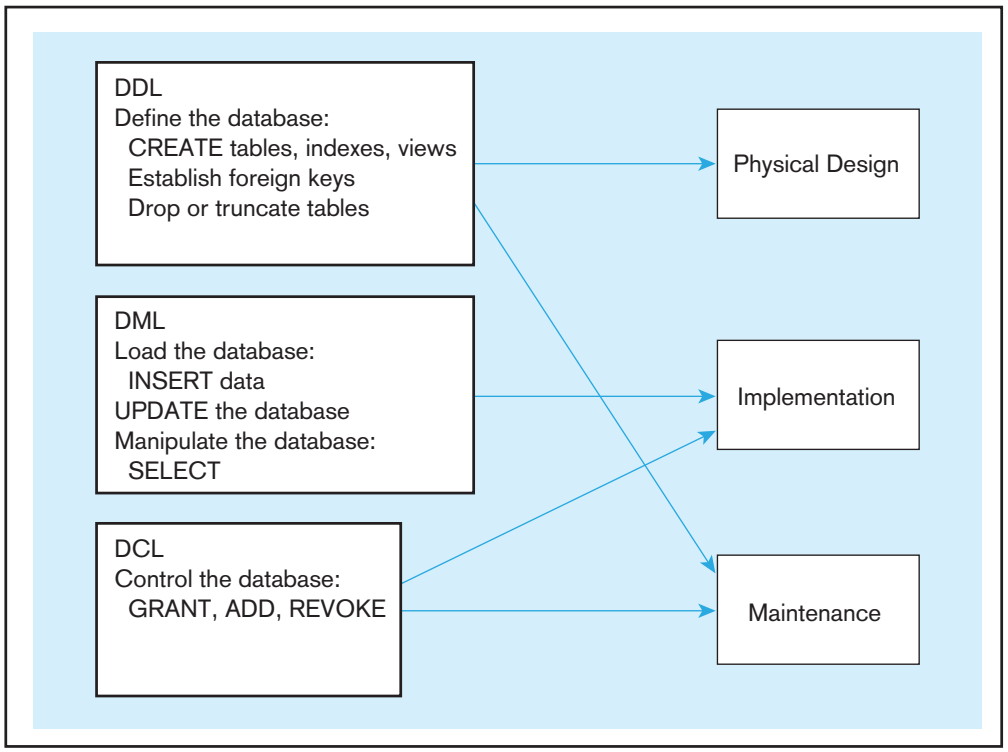


FIGURE 6-4 DDL, DML, DCL, and the database development process

5. Each SQL command ends with a semicolon (;). In interactive mode, when the user presses Enter, the SQL command will execute. Be alert for alternate conventions, such as typing GO or having to include a continuation symbol such as a hyphen at the end of each line used in the command. The spacing and indentations shown here are included for readability and are not a required part of standard SQL syntax.

DEFINING A DATABASE IN SQL

Because most systems allocate storage space to contain base tables, views, constraints, indexes, and other database objects when a database is created, you may not be allowed to create a database. Because of this, the privilege of creating databases may be reserved for the database administrator, and you may need to ask to have a database created. Students at a university may be assigned an account that gives access to an existing database, or they may be allowed to create their own database in a limited amount of allocated storage space (sometimes called *perm space* or *table space*). In any case, the basic syntax for creating a database is

```
CREATE SCHEMA database_name; AUTHORIZATION owner_user id
```

The database will be owned by the authorized user, although it is possible for other specified users to work with the database or even to transfer ownership of the database. Physical storage of the database is dependent on both the hardware and software environment and is usually the concern of the system administrator. The amount of control over physical storage that a database administrator is able to exert depends on the RDBMS being used. Little control is possible when using Microsoft Access, but Microsoft SQL Server 2008 allows for more control of the physical database. A database administrator may exert considerable control over the placement of data, control files, index files, schema ownership, and so forth, thus improving the ability to tune the database to perform more efficiently and to create a secure database environment.

Generating SQL Database Definitions

Several SQL DDL CREATE commands are included in SQL:200n (and each command is followed by the name of the object being created):

CREATE SCHEMA	Used to define the portion of a database that a particular user owns. Schemas are dependent on a catalog and contain schema objects, including base tables and views, domains, constraints, assertions, character sets, collations, and so forth.
CREATE TABLE	Defines a new table and its columns. The table may be a base table or a derived table. Tables are dependent on a schema. Derived tables are created by executing a query that uses one or more tables or views.
CREATE VIEW	Defines a logical table from one or more tables or views. Views may not be indexed. There are limitations on updating data through a view. Where views can be updated, those changes can be transferred to the underlying base tables originally referenced to create the view.

You don't have to be perfect when you create these objects, and they don't have to last forever. Each of these CREATE commands can be reversed by using a DROP command. Thus, DROP TABLE *tablename* will destroy a table, including its definition, contents, and any constraints, views, or indexes associated with it. Usually only the table creator may delete the table. DROP SCHEMA or DROP VIEW will also destroy the named schema or view. ALTER TABLE may be used to change the definition of an existing base table by adding, dropping, or changing a column or by dropping a constraint. Some RDBMSs will not allow you to alter a table in a way that the current data in that table will violate the new definitions (e.g., you cannot create a new constraint when current data will violate that constraint, or if you change the precision of a numeric column you may lose the extra precision of more precise existing values).

There are also five other CREATE commands included in the SQL standards; we list them here but do not cover them in this book:

CREATE CHARACTER SET	Allows the user to define a character set for text strings and aids in the globalization of SQL by enabling the use of languages other than English. Each character set contains a set of characters, a way to represent each character internally, a data format used for this representation, and a collation, or way of sorting the character set.
CREATE COLLATION	A named schema object that specifies the order that a character set will assume. Existing collations may be manipulated to create a new collation.
CREATE TRANSLATION	A named set of rules that maps characters from a source character set to a destination character set for translation or conversion purposes.
CREATE ASSERTION	A schema object that establishes a CHECK constraint that is violated if the constraint is false.
CREATE DOMAIN	A schema object that establishes a domain, or set of valid values, for an attribute. Data type will be specified, and a default value, collation, or other constraint may also be specified, if desired.

Creating Tables

Once the data model is designed and normalized, the columns needed for each table can be defined, using the SQL CREATE TABLE command. The general syntax for CREATE TABLE is shown in Figure 6-5. Here is a series of steps to follow when preparing to create a table:

1. Identify the appropriate data type, including length, precision, and scale, if required, for each attribute.
2. Identify the columns that should accept null values, as discussed in Chapter 5. Column controls that indicate a column cannot be null are established when a

```
CREATE TABLE tablename
( {column definition       [table constraint] } . , . .
[ON COMMIT {DELETE | P RESERVE} ROWS] );

where column definition ::=
column_name
        {domain namel d atatype [(size)] }
        [column_constraint_clause. . .]
        [default value]
        [collate clause]

and table constraint ::=
        [CONSTRAINT constraint_name]
        Constraint_type [constraint_attributes]
```

FIGURE 6-5 General syntax of the CREATE TABLE statement used in data definition language

table is created and are enforced for every update of the table when data are entered.

3. Identify the columns that need to be unique. When a column control of UNIQUE is established for a column, the data in that column must have a different value for each row of data within that table (i.e., no duplicate values). Where a column or set of columns is designated as UNIQUE, that column or set of columns is a candidate key, as discussed in Chapter 4. Although each base table may have multiple candidate keys, only one candidate key may be designated as a PRIMARY KEY. When a column(s) is specified as the PRIMARY KEY, that column(s) is also assumed to be NOT NULL, even if NOT NULL is not explicitly stated. UNIQUE and PRIMARY KEY are both column constraints. Note that a table with a composite primary key, OrderLine_T, is defined in Figure 6-6. The OrderLine_PK constraint includes both OrderID and ProductID in the primary key constraint, thus creating a composite key. Additional attributes may be included within the parentheses as needed to create the composite key.
4. Identify all primary key–foreign key mates, as presented in Chapter 4. Foreign keys can be established immediately, as a table is created, or later by altering the table. The parent table in such a parent–child relationship should be created first so that the child table will reference an existing parent table when it is created. The column constraint REFERENCES can be used to enforce referential integrity (e.g., the Order_FK constraint on the Order_T table).
5. Determine values to be inserted in any columns for which a default value is desired. DEFAULT can be used to define a value that is automatically inserted when no value is inserted during data entry. In Figure 6-6, the command that creates the Order_T table has defined a default value of SYSDATE (Oracle's name for the current date) for the OrderDate attribute.
6. Identify any columns for which domain specifications may be stated that are more constrained than those established by data type. Using CHECK as a column constraint, it may be possible to establish validation rules for values to be inserted into the database. In Figure 6-6, creation of the Product_T table includes a check constraint, which lists the possible values for Product_Finish. Thus, even though an entry of 'White Maple' would meet the VARCHAR data type constraints, it would be rejected because 'White Maple' is not in the checklist.
7. Create the table and any desired indexes, using the CREATE TABLE and CREATE INDEX statements. (CREATE INDEX is not a part of the SQL:1999 standard because indexing is used to address performance issues, but it is available in most RDBMSs.)

Figure 6-6 shows database definition commands using Oracle 11g that include additional column constraints, as well as primary and foreign keys given names. For example, the Customer table's primary key is CustomerID. The primary key constraint is named Customer_PK. In Oracle, for example, once a constraint has been given a

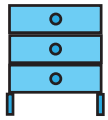


FIGURE 6-6 SQL database definition commands for Pine Valley Furniture Company (Oracle 11g)

```
CREATE TABLE Customer_T
                    (CustomerID                    NUMBER(11,0)      NOT NULL,
                     CustomerName                  VARCHAR2(25)      NOT NULL,
                     CustomerAddress               VARCHAR2(30),
                     CustomerCity                  VARCHAR2(20),
                     CustomerState                 CHAR(2),
                     CustomerPostalCode            VARCHAR2(9),
CONSTRAINT Customer_PK PRIMARY KEY (CustomerID));

CREATE TABLE Order_T
                   (OrderID                        NUMBER(11,0)      NOT NULL,
                    OrderDate                      DATE DEFAULT SYSDATE,
                    CustomerID                     NUMBER(11,0),
CONSTRAINT Order_PK PRIMARY KEY (OrderID),
CONSTRAINT Order_FK FOREIGN KEY (CustomerID) REFERENCES Customer_T(CustomerID));

CREATE TABLE Product_T
                   (ProductID                      NUMBER(11,0)      NOT NULL,
                    ProductDescription             VARCHAR2(50),
                    ProductFinish                  VARCHAR2(20)
                                  CHECK (ProductFinish IN ('Cherry', 'Natural Ash', 'White Ash',
                                                           'Red Oak', 'Natural Oak', 'Walnut')),
                    ProductStandardPrice           DECIMAL(6,2),
                    ProductLineID                  INTEGER,
CONSTRAINT Product_PK PRIMARY KEY (ProductID));

CREATE TABLE OrderLine_T
                   (OrderID                        NUMBER(11,0)      NOT NULL,
                    ProductID                      INTEGER           NOT NULL,
                    OrderedQuantity                NUMBER(11,0),
CONSTRAINT OrderLine_PK PRIMARY KEY (OrderID, ProductID),
CONSTRAINT OrderLine_FK1 FOREIGN KEY (OrderID) REFERENCES Order_T(OrderID),
CONSTRAINT OrderLine_FK2 FOREIGN KEY (ProductID) REFERENCES Product_T(ProductID));
```

meaningful name by the user, a database administrator will find it easy to identify the primary key constraint on the customer table because its name, Customer_PK, will be the value of the constraint_name column in the DBA_CONSTRAINTS table. If a meaningful constraint name were not assigned, a 16-byte system identifier would be assigned automatically. These identifiers are difficult to read and even more difficult to match up with user-defined constraints. Documentation about how system identifiers are generated is not available, and the method can be changed without notification. Bottom line: Give all constraints names or be prepared for extra work later.

When a foreign key constraint is defined, referential integrity will be enforced. This is good: We want to enforce business rules in the database. Fortunately, you are still allowed to have a null value for the foreign key (signifying a zero cardinality of the relationship) as long as you do not put the NOT NULL clause on the foreign key column. For example, if you try to add an order with an invalid CustomerID value (every order has to be related to some customer, so the minimum cardinality is one next to Customer for the Submits relationship in Figure 2-22), you will receive an error message. Each DBMS vendor generates its own error messages, and these messages may be difficult to interpret. Microsoft Access, being intended for both personal and professional use, provides simple error messages in dialog boxes. For example, for a referential integrity violation, Access displays the following error message: "You cannot add or change a record because a related record is required in table Customer_T." No record will be added to Order_T until that record references an existing customer in the Customer_T table.

Sometimes a user will want to create a table that is similar to one that already exists. SQL:1999 included the capability of adding a LIKE clause to the CREATE TABLE statement to allow for the copying of the existing structure of one or more tables into a

new table. For example, a table can be used to store data that are questionable until the questionable data can be reviewed by an administrator. This exception table has the same structure as the verified transaction table, and missing or conflicting data will be reviewed and resolved before those transactions are appended to the transaction table. SQL:200n has expanded the CREATE . . . LIKE capability by allowing additional information, such as table constraints, from the original table to be easily ported to the new table when it is created. The new table exists independently of the original table. Inserting a new instance into the original table will have no effect on the new table. However, if the attempt to insert the new instance triggers an exception, the trigger can be written so that the data is stored in the new table to be reviewed later.

Oracle, MySQL, and some other RDBMSs have an interesting "dummy" table that is automatically defined with each database—the Dual table. The Dual table is used to run an SQL command against a system variable. For example,

```
SELECT Sysdate FROM Dual;
```

displays the current date, and

```
SELECT 8 + 4 FROM Dual;
```

displays the result of this arithmetic.

Creating Data Integrity Controls

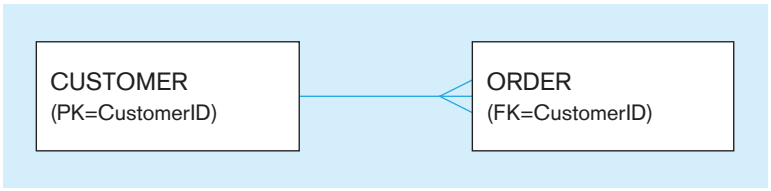
We have seen the syntax that establishes foreign keys in Figure 6-6. To establish referential integrity constraint between two tables with a 1:*M* relationship in the relational data model, the primary key of the table on the one side will be referenced by a column in the table on the many side of the relationship. Referential integrity means that a value in the matching column on the many side must correspond to a value in the primary key for some row in the table on the one side or be NULL. The SQL REFERENCES clause prevents a foreign key value from being added if it is not already a valid value in the referenced primary key column, but there are other integrity issues.

If a CustomerID value is changed, the connection between that customer and orders placed by that customer will be ruined. The REFERENCES clause prevents making such a change in the foreign key value, but not in the primary key value. This problem could be handled by asserting that primary key values cannot be changed once they are established. In this case, updates to the customer table will be handled in most systems by including an ON UPDATE RESTRICT clause. Then, any updates that would delete or change a primary key value will be rejected unless no foreign key references that value in any child table. See Figure 6-7 for the syntax associated with updates.

Another solution is to pass the change through to the child table(s) by using the ON UPDATE CASCADE option. Then, if a customer ID number is changed, that change will flow through (cascade) to the child table, Order_T, and the customer's ID will also be updated in the Order_T table.

A third solution is to allow the update on Customer_T but to change the involved CustomerID value in the Order_T table to NULL by using the ON UPDATE SET NULL option. In this case, using the SET NULL option would result in losing the connection between the order and the customer, which is not a desired effect. The most flexible option to use would be the CASCADE option. If a customer record were deleted, ON DELETE RESTRICT, CASCADE, or SET NULL would also be available. With DELETE RESTRICT, the customer record could not be deleted unless there were no orders from that customer in the Order_T table. With DELETE CASCADE, removing the customer would remove all associated order records from Order_T. With DELETE SET NULL, the order records for that customer would be set to null before the customer's record was deleted. With DELETE SET DEFAULT, the order records for that customer would be set to a default value before the customer's record was deleted. DELETE RESTRICT would probably make the most sense. Not all SQL RDBMSs provide for primary key referential integrity. In that case, update and delete permissions on the primary key column may be revoked.

FIGURE 6-7 **Ensuring data integrity through updates**



Restricted Update: A customer ID can only be deleted if it is not found in ORDER table.

```
CREATE TABLE CustomerT
                (CustomerID          INTEGER DEFAULT '999'     NOT NULL,
                 CustomerName        VARCHAR(40)               NOT NULL,
                 . . .
CONSTRAINT Customer_PK PRIMARY KEY (CustomerID),
ON UPDATE RESTRICT);
```

Cascaded Update: Changing a customer ID in the CUSTOMER table will result in that value changing in the ORDER table to match.

```
. . . ON UPDATE CASCADE);
```

Set Null Update: When a customer ID is changed, any customer ID in the ORDER table that matches the old customer ID is set to NULL.

```
. . . ON UPDATE SET NULL);
```

Set Default Update: When a customer ID is changed, any customer ID in the ORDER tables that matches the old customer ID is set to a predefined default value.

```
. . . ON UPDATE SET DEFAULT);
```

Changing Table Definitions

Base table definitions may be changed by using ALTER on the column specifications. The ALTER TABLE command can be used to add new columns to an existing table. Existing columns may also be altered. Table constraints may be added or dropped. The ALTER TABLE command may include keywords such as ADD, DROP, or ALTER and allow the column's names, data type, length, and constraints to be changed. Usually, when adding a new column, its null status will be NULL so that data that have already been entered in the table can be dealt with. When the new column is created, it is added to all of the instances in the table, and a value of NULL would be the most reasonable. The ALTER command cannot be used to change a view.

Syntax:

```
ALTER TABLE table_name alter_table_action;
```

Some of the alter_table_actions available are:

```
ADD [COLUMN] column_definition
ALTER [COLUMN] column_name SET DEFAULT default-value
ALTER [COLUMN] column_name DROP DEFAULT
DROP [COLUMN] column_name [RESTRICT] [CASCADE]
ADD table_constraint
```

Command: To add a customer type column named CustomerType to the Customer table.

```
ALTER TABLE CUSTOMER_T
ADD COLUMN CustomerType VARCHAR2 (2) DEFAULT "Commercial";
```

The ALTER command is invaluable for adapting a database to inevitable modifications due to changing requirements, prototyping, evolutionary development, and

mistakes. It is also useful when performing a bulk data load into a table that contains a foreign key. The constraint may be temporarily dropped. Later, after the bulk data load has finished, the constraint can be enabled. When the constraint is reenabled, it is possible to generate a log of any records that have referential integrity problems. Rather than have the data load balk each time such a problem occurs during the bulk load, the database administrator can simply review the log and reconcile the few (hopefully few) records that were problematic.

Removing Tables

To remove a table from a database, the owner of the table may use the DROP TABLE command. Views are dropped by using the similar DROP VIEW command.

Command: To drop a table from a database schema.

```
DROP TABLE Customer_T;
```

This command will drop the table and save any pending changes to the database. To drop a table, you must either own the table or have been granted the DROP ANY TABLE system privilege. Dropping a table will also cause associated indexes and privileges granted to be dropped. The DROP TABLE command can be qualified by the keywords RESTRICT or CASCADE. If RESTRICT is specified, the command will fail, and the table will not be dropped if there are any dependent objects, such as views or constraints, that currently reference the table. If CASCADE is specified, all dependent objects will also be dropped as the table is dropped. Many RDBMSs allows users to retain the table's structure but remove all of the data that have been entered in the table with its TRUNCATE TABLE command. Commands for updating and deleting part of the data in a table are covered in the next section.

INSERTING, UPDATING, AND DELETING DATA

Once tables have been created, it is necessary to populate them with data and maintain those data before queries can be written. The SQL command that is used to populate tables is the INSERT command. When entering a value for every column in the table, you can use a command like the following, which was used to add the first row of data to the Customer_T table for Pine Valley Furniture Company. Notice that the data values must be ordered in the same order as the columns in the table.

Command: To insert a row of data into a table where a value will be inserted for every attribute.

```
INSERT INTO Customer_T VALUES
(001, 'Contemporary Casuals', '1355 S. Himes Blvd.', 'Gainesville', 'FL', 32601);
```

When data will not be entered into every column in the table, either enter the value NULL for the empty fields or specify those columns to which data are to be added. Here, too, the data values must be in the same order as the columns have been specified in the INSERT command. For example, the following statement was used to insert one row of data into the Product_T table, because there was no product line ID for the end table.

Command: To insert a row of data into a table where some attributes will be left null.

```
INSERT INTO Product_T (ProductID,
ProductDescription, ProductFinish, ProductStandardPrice)
    VALUES (1, 'End Table', 'Cherry', 175, 8);
```

In general, the INSERT command places a new row in a table, based on values supplied in the statement, copies one or more rows derived from other database data into a table, or extracts data from one table and inserts them into another. If you want to populate a table, CaCustomer_T, that has the same structure as CUSTOMER_T,

with only Pine Valley's California customers, you could use the following INSERT command.

Command: Populating a table by using a subset of another table with the same structure.

```
INSERT INTO CaCustomer_T
SELECT * FROM Customer_T
    WHERE CustomerState = 'CA';
```

In many cases, we want to generate a unique primary identifier or primary key every time a row is added to a table. Customer identification numbers are a good example of a situation where this capability would be helpful. SQL:200n had added a new feature, identity columns, that removes the previous need to create a procedure to generate a sequence and then apply it to the insertion of data. To take advantage of this, the CREATE TABLE Customer_T statement displayed in Figure 6-6 may be modified (emphasized by bold print) as follows:

```
CREATE TABLE Customer_T
(CustomerID INTEGER GENERATED ALWAYS AS IDENTITY
    (START WITH 1
    INCREMENT BY 1
    MINVALUE 1
    MAXVALUE 10000
    NO CYCLE),
CustomerName          VARCHAR2(25) NOT NULL,
CustomerAddress       VARCHAR2(30),
CustomerCity          VARCHAR2(20),
CustomerState         CHAR(2),
CustomerPostalCode    VARCHAR2(9),
CONSTRAINT Customer_PK PRIMARY KEY (CustomerID);
```

Only one column can be an identity column in a table. When a new customer is added, the CustomerID value will be assigned implicitly if the vendor has implemented identity columns.

Thus, the command that adds a new customer to Customer_T will change from this:

```
INSERT INTO Customer_T VALUES
(001, 'Contemporary Casuals', '1355 S. Himes Blvd.', 'Gainesville',
  'FL', 32601);
```

to this:

```
INSERT INTO Customer_T VALUES
('Contemporary Casuals', '1355 S. Himes Blvd.', 'Gainesville', 'FL', 32601);
```

The primary key value, 001, does not need to be entered, and the syntax to accomplish the automatic sequencing has been simplified in SQL:200n.

Batch Input

The INSERT command is used to enter one row of data at a time or to add multiple rows as the result of a query. Some versions of SQL have a special command or utility for entering multiple rows of data as a batch: the INPUT command. For example, Oracle includes a program, SQL*Loader, which runs from the command line and can be used to load data from a file into the database. SQL Server includes a BULK INSERT command with Transact-SQL for importing data into a table or view. (These powerful and feature rich programs are not within the scope of this text.)

Deleting Database Contents

Rows can be deleted from a database individually or in groups. Suppose Pine Valley Furniture decides that it will no longer deal with customers located in Hawaii. Customer_T rows for customers with addresses in Hawaii could all be eliminated using the next command.

Command: Deleting rows that meet a certain criterion from the Customer table.

```
DELETE FROM Customer_T
WHERE CustomerState = 'HI';
```

The simplest form of DELETE eliminates all rows of a table.

Command: Deleting all rows from the Customer table.

```
DELETE FROM Customer_T;
```

This form of the command should be used very carefully!

Deletion must also be done with care when rows from several relations are involved. For example, if we delete a Customer_T row, as in the previous query, before deleting associated Order_T rows, we will have a referential integrity violation, and the DELETE command will not execute. (Note: Including the ON DELETE clause with a field definition can mitigate such a problem. Refer to the "Creating Data Integrity Controls" section in this chapter if you've forgotten about the ON clause.) SQL will actually eliminate the records selected by a DELETE command. Therefore, always execute a SELECT command first to display the records that would be deleted and visually verify that only the desired rows are included.

Updating Database Contents

To update data in SQL, we must inform the DBMS what relation, columns, and rows are involved. If an incorrect price is entered for the dining table in the Product_T table, the following SQL UPDATE statement would establish the correction.

Command: To modify standard price of product 7 in the Product table to 775.

```
UPDATE Product_T
SET ProductStandardPrice = 775
    WHERE ProductID = 7;
```

The SET command can also change a value to NULL; the syntax is SET column-name = NULL. As with DELETE, the WHERE clause in an UPDATE command may contain a subquery, but the table being updated may not be referenced in the subquery. Subqueries are discussed in Chapter 7.

The SQL:200n standard has included a new keyword, MERGE, that makes updating a table easier. Many database applications need to update master tables with new data. A Purchases_T table, for example, might include rows with data about new products and rows that change the standard price of existing products. Updating Product_T can be accomplished by using INSERT to add the new products and UPDATE to modify StandardPrice in an SQL-92 or SQL:1999 DBMS. SQL:200n compliant DBMSs can accomplish the update and the insert in one step by using MERGE:

```
MERGE INTO Product_T AS PROD
USING
(SELECT ProductID, ProductDescription, ProductFinish,
ProductStandardPrice, ProductLineID FROM Purchases_T) AS PURCH
    ON (PROD.ProductID = PURCH.ProductID)
WHEN MATCHED THEN UPDATE
    PROD.ProductStandardPrice = PURCH.ProductStandardPrice
```

```
WHEN NOT MATCHED THEN INSERT
    (ProductID, ProductDescription, ProductFinish, ProductStandardPrice,
    ProductLineID)
    VALUES(PURCH.ProductID, PURCH.ProductDescription,
    PURCH.ProductFinish, PURCH.ProductStandardPrice,
      PURCH.ProductLineID);
```

INTERNAL SCHEMA DEFINITION IN RDBMSS

The internal schema of a relational database can be controlled for processing and storage efficiency. The following are some techniques used for tuning the operational performance of the relational database internal data model:

1. Choosing to index primary and/or secondary keys to increase the speed of row selection, table joining, and row ordering. You can also drop indexes to increase speed of table updating. You may want to review the section in Chapter 5 on selecting indexes.
2. Selecting file organizations for base tables that match the type of processing activity on those tables (e.g., keeping a table physically sorted by a frequently used reporting sort key).
3. Selecting file organizations for indexes, which are also tables, appropriate to the way the indexes are used and allocating extra space for an index file so that an index can grow without having to be reorganized.
4. Clustering data so that related rows of frequently joined tables are stored close together in secondary storage to minimize retrieval time.
5. Maintaining statistics about tables and their indexes so that the DBMS can find the most efficient ways to perform various database operations.

Not all of these techniques are available in all SQL systems. Indexing and clustering are typically available, however, so we discuss these in the following sections.

Creating Indexes

Indexes are created in most RDBMSs to provide rapid random and sequential access to base-table data. Because the ISO SQL standards do not generally address performance issues, no standard syntax for creating indexes is included. The examples given here use Oracle syntax and give a feel for how indexes are handled in most RDBMSs. Note that although users do not directly refer to indexes when writing any SQL command, the DBMS recognizes which existing indexes would improve query performance. Indexes can usually be created for both primary and secondary keys and both single and concatenated (multiple-column) keys. In some systems, users can choose between ascending and descending sequences for the keys in an index.

For example, an alphabetical index on CustomerName in the Customer_T table in Oracle is created here.

Command: To create an alphabetical index on customer name in the Customer table.

```
CREATE INDEX Name_IDX ON Customer_T (CustomerName);
```

RDBMs usually support several different types of indexes, each of which assists in different kinds of keyword searches. For example, in MySQL you can create unique (appropriate for primary keys), nonunique (secondary keys), fulltext (used for full-text searches), spatial (used for spatial data types), and hash (which is used for in-memory tables).

Indexes can be created or dropped at any time. If data already exist in the key column(s), index population will automatically occur for the existing data. If an index is defined as UNIQUE (using the syntax CREATE UNIQUE INDEX . . .) and the existing data violate this condition, the index creation will fail. Once an index is created, it will be updated as data are entered, updated, or deleted.

When we no longer need tables, views, or indexes, we use the associated DROP statements. For example, the NAME_IDX index from the previous example is dropped here.

Command: To remove the index on the customer name in the Customer table.

```
DROP INDEX Name_IDX;
```

Although it is possible to index every column in a table, use caution when deciding to create a new index. Each index consumes extra storage space and also requires overhead maintenance time whenever indexed data change value. Together, these costs may noticeably slow retrieval response times and cause annoying delays for online users. A system may use only one index even if several are available for keys in a complex qualification. A database designer must know exactly how indexes are used by the particular RDBMS in order to make wise choices about indexing. Oracle includes an explain plan tool that can be used to look at the order in which an SQL statement will be processed and at the indexes that will be used. The output also includes a cost estimate that can be compared with estimates from running the statement with different indexes to determine which is most efficient.

PROCESSING SINGLE TABLES

"Processing single tables" may seem like Friday night at the hottest club in town, but we have something else in mind. Sorry, no dating suggestions (and sorry for the pun).

Four data manipulation language commands are used in SQL. We have talked briefly about three of them (UPDATE, INSERT, and DELETE) and have seen several examples of the fourth, SELECT. Although the UPDATE, INSERT, and DELETE commands allow modification of the data in the tables, it is the SELECT command, with its various clauses, that allows users to query the data contained in the tables and ask many different questions or create ad hoc queries. The basic construction of an SQL command is fairly simple and easy to learn. Don't let that fool you; SQL is a powerful tool that enables users to specify complex data analysis processes. However, because the basic syntax is relatively easy to learn, it is also easy to write SELECT queries that are syntactically correct but do not answer the exact question that is intended. Before running queries against a large production database, always test them carefully on a small test set of data to be sure that they are returning the correct results. In addition to checking the query results manually, it is often possible to parse queries into smaller parts, examine the results of these simpler queries, and then recombine them. This will ensure that they act together in the expected way. We begin by exploring SQL queries that affect only a single table. In Chapter 7, we join tables and use queries that require more than one table.

Clauses of the SELECT Statement

Most SQL data retrieval statements include the following three clauses:

SELECT	Lists the columns (including expressions involving columns) from base tables, derived tables, or views to be projected into the table that will be the result of the command. (That's the technical way of saying it lists the data you want to display.)
FROM	Identifies the tables, derived tables, or views from which columns will be chosen to appear in the result table and includes the tables, derived tables, or views needed to join tables to process the query.
WHERE	Includes the conditions for row selection within the items in the FROM clause and the conditions between tables, derived tables, or views for joining. Because SQL is considered a set manipulation language, the WHERE clause is important in defining the set of rows being manipulated.

The first two clauses are required, and the third is necessary when only certain table rows are to be retrieved or multiple tables are to be joined. (Most examples for this section are drawn from the data shown in Figure 6-3.) For example, we can display product name and quantity on hand from the PRODUCT view for all Pine Valley Furniture Company products that have a standard price of less than $275.

Query: Which products have a standard price of less than $275?

```
SELECT ProductDescription, ProductStandardPrice
  FROM Product_T
    WHERE ProductStandardPrice < 275;
```

Result:

PRODUCTDESCRIPTION	PRODUCTSTANDARDPRICE
End Table	175
Computer Desk	250
Coffee Table	200

As stated before, in this book we show results (except where noted) in the style of Oracle, which means that column headings are in all capital letters. If this is too annoying for users, then the data names should be defined with an underscore between the words rather than run-on words, or you can use an alias (described later in this section) to redefine a column heading for display.

Every SELECT statement returns a result table (a set of rows) when it executes. So, SQL is consistent—tables in, tables out of every query. This becomes important with more complex queries because we can use the result of one query (a table) as part of another query (e.g., we can include a SELECT statement as one of the elements in the FROM clause, creating a derived table, which we illustrate later in this chapter).

Two special keywords can be used along with the list of columns to display: DISTINCT and *. If the user does not wish to see duplicate rows in the result, SELECT DISTINCT may be used. In the preceding example, if the other computer desk carried by Pine Valley Furniture had also cost less than $275, the results of the query would have had duplicate rows. SELECT DISTINCT ProductDescription would display a result table without the duplicate rows. SELECT *, where * is used as a wildcard to indicate all columns, displays all columns from all the items in the FROM clause.

Also, note that the clauses of a SELECT statement must be kept in order, or syntax error messages will occur and the query will not execute. It may also be necessary to qualify the names of the database objects according to the SQL version being used. If there is any ambiguity in an SQL command, you must indicate exactly from which table, derived table, or view the requested data are to come. For example, in Figure 6-3 CustomerID is a column in both Customer_T and Order_T. When you own the database being used (i.e., the user created the tables) and you want CustomerID to come from Customer_T, specify it by asking for Customer_T.CustomerID. If you want CustomerID to come from Order_T, then ask for Order_T.CustomerID. Even if you don't care which table CustomerID comes from, it must be specified because SQL can't resolve the ambiguity without user direction. When you are allowed to use data created by someone else, you must also specify the owner of the table by adding the owner's user ID. Now a request to SELECT the CustomerID from Customer_T may look like this: OWNER_ID.Customer_T.CustomerID. The examples in this book assume that the reader owns the tables or views being used, as the SELECT statements will be easier to read without the qualifiers. Qualifiers will be included where necessary and may always be included in statements if desired. Problems may occur when qualifiers are left out, but no problems will occur when they are included.

If typing the qualifiers and column names is wearisome (computer keyboards aren't, yet, built to accommodate the two-thumb cellphone texting technique), or if the column names will not be meaningful to those who are reading the reports, establish aliases for data names that will then be used for the rest of the query. Although SQL:1999 does not include aliases or synonyms, they are widely implemented and aid in readability and simplicity in query construction.

Query: What is the address of the customer named Home Furnishings? Use an alias, Name, for the customer name. (The AS clauses are bolded for emphasis only.)

```
SELECT CUST.CustomerName AS Name, CUST.CustomerAddress
  FROM ownerid.Customer_T AS Cust
    WHERE Name = 'Home Furnishings';
```

This retrieval statement will give the following result in many versions of SQL. In Oracle's SQL*Plus, the alias for the column cannot be used in the rest of the SELECT statement, except in a HAVING clause, so in order for the query to run, CustomerName would have to be used in the last line rather than Name. Notice that the column header prints as Name rather than CustomerName and that the table alias may be used in the SELECT clause even though it is not defined until the FROM clause.

Result:

NAME	CUSTOMERADDRESS
Home Furnishings	1900 Allard Ave.

You've likely concluded that SQL generates pretty plain output. Using an alias is a good way to make column headings more readable. (Aliases also have other uses, which we'll address later.) Many RDBMSs have other proprietary SQL clauses to improve the display of data. For example, Oracle has the COLUMN clause of the SELECT statement, which can be used to change the text for the column heading, change alignment of the column heading, reformat the column value, or control wrapping of data in a column, among other properties. You may want to investigate such capabilities for the RDBMS you are using.

When you use the SELECT clause to pick out the columns for a result table, the columns can be rearranged so that they will be ordered differently in the result table than in the original table. In fact, they will be displayed in the same order as they are included in the SELECT statement. Look back at Product_T in Figure 6-3 to see the different ordering of the base table from the result table for this query.

Query: List the unit price, product name, and product ID for all products in the Product table.

```
SELECT ProductStandardPrice, ProductDescription, ProductID
FROM Product_T;
```

Result:

PRODUCTSTANDARDPRICE	PRODUCTDESCRIPTION	PRODUCTID
175	End Table	1
200	Coffee Table	2
375	Computer Desk	3
650	Entertainment Center	4
325	Writer's Desk	5
750	8-Drawer Desk	6
800	Dining Table	7
250	Computer Desk	8

Using Expressions

The basic SELECT . . . FROM . . . WHERE clauses can be used with a single table in a number of ways. You can create expressions, which are mathematical manipulations of the data in the table, or take advantage of stored functions, such as SUM or AVG, to manipulate the chosen rows of data from the table. Mathematical manipulations can be constructed by using the + for addition, – for subtraction, * for multiplication, and / for division. These operators can be used with any numeric columns. Expressions are computed for each row of the result table, such as displaying the difference between the standard price and unit cost of a product, or they can involve computations of columns

and functions, such as standard price of a product multiplied by the amount of that product sold on a particular order (which would require summing OrderedQuantities). Some systems also have an operand called modulo, usually indicated by %. A modulo is the integer remainder that results from dividing two integers. For example, 14 % 4 is 2 because 14/4 is 3, with a remainder of 2. The SQL standard supports year–month and day–time intervals, which make it possible to perform date and time arithmetic (e.g., to calculate someone's age from today's date and a person's birthday).

Perhaps you would like to know the current standard price of each product and its future price if all prices were increased by 10 percent. Using SQL*Plus, here are the query and the results.

Query: What are the standard price and standard price if increased by 10 percent for every product?

```
SELECT ProductID, ProductStandardPrice, ProductStandardPrice*1.1 AS
Plus10Percent
  FROM Product_T;
```

Result:

PRODUCTID	PRODUCTSTANDARDPRICE	PLUS10PERCENT
2	200.0000	220.00000
3	375.0000	412.50000
1	175.0000	192.50000
8	250.0000	275.00000
7	800.0000	880.00000
5	325.0000	357.50000
4	650.0000	715.00000
6	750.0000	825.00000

The precedence rules for the order in which complex expressions are evaluated are the same as those used in other programming languages and in algebra. Expressions in parentheses will be calculated first. When parentheses do not establish order, multiplication and division will be completed first, from left to right, followed by addition and subtraction, also left to right. To avoid confusion, use parentheses to establish order. Where parentheses are nested, the innermost calculations will be completed first.

Using Functions

Standard SQL identifies a wide variety of mathematical, string and date manipulation, and other functions. We will illustrate some of the mathematical functions in this section. You will want to investigate what functions are available with the DBMS you are using, some of which may be proprietary to that DBMS. The standard functions include the following:

Mathematical	MIN, MAX, COUNT, SUM, ROUND (to round up a number to a specific number of decimal places), TRUNC (to truncate insignificant digits), and MOD (for modular arithmetic)
String	LOWER (to change to all lower case), UPPER (to change to all capital letters), INITCAP (to change to only an initial capital letter), CONCAT (to concatenate), SUBSTR (to isolate certain character positions), and COALESCE (finding the first not NULL values in a list of columns)
Date	NEXT_DAY (to compute the next date in sequence), ADD_MONTHS (to compute a date a given number of months before or after a given date), and MONTHS_BETWEEN (to compute the number of months between specified dates)
Analytical	TOP (find the top *n* values in a set, e.g., the top 5 customers by total annual sales)

Perhaps you want to know the average standard price of all inventory items. To get the overall average value, use the AVG stored function. We can name the resulting expression with an alias, AveragePrice. Using SQL*Plus, here are the query and the results.

Query: What is the average standard price for all products in inventory?

```
SELECT AVG (ProductStandardPrice) AS AveragePrice
FROM Product_T;
```

Result:

```
AVERAGEPRICE
------------
      440.625
```

SQL:1999 stored functions include ANY, AVG, COUNT, EVERY, GROUPING, MAX, MIN, SOME, and SUM. SQL:200n adds LN, EXP, POWER, SQRT, FLOOR, CEILING, and WIDTH_BUCKET. New functions tend to be added with each new SQL standard, and more functions have been added in SQL:2003 and SQL:2008, many of which are for advanced analytical processing of data (e.g., calculating moving averages and statistical sampling of data). As seen in the above example, functions such as COUNT, MIN, MAX, SUM, and AVG of specified columns in the column list of a SELECT command may be used to specify that the resulting answer table is to contain aggregated data instead of row-level data. Using any of these aggregate functions will give a one-row answer.

Query: How many different items were ordered on order number 1004?

```
SELECT COUNT (*)
  FROM OrderLine_T
    WHERE OrderID = 1004;
```

Result:

```
COUNT (*)
---------
        2
```

It seems that it would be simple enough to list order number 1004 by changing the query.

Query: How many different items were ordered on order number 1004, and what are they?

```
SELECT ProductID, COUNT (*)
  FROM OrderLine_T
    WHERE OrderID = 1004;
```

In Oracle, here is the result.

Result:

```
ERROR at line 1:
ORA-00937: not a single-group group function
```

And in Microsoft SQL Server, the result is as follows.

Result:

```
Column 'OrderLine_T.ProductID' is invalid in the select list because
it is not contained in an Aggregate function and there is no
GROUP BY clause.
```

The problem is that ProductID returns two values, 6 and 8, for the two rows selected, whereas COUNT returns one aggregate value, 2, for the set of rows with ID = 1004. In most implementations, SQL cannot return both a row value and a set value; users must run two separate queries, one that returns row information and one that returns set information.

A similar issue arises if we try to find the difference between the standard price of each product and the overall average standard price (which we calculated above). You might think the query would be

```
SELECT ProductStandardPrice – AVG(ProductStandardPrice)
  FROM Product_T;
```

However, again we have mixed a column value with an aggregate, which will cause an error. Remember that the FROM list can contain tables, derived tables, and views. One approach to developing a correct query is to make the aggregate the result of a derived table, as we do in the following sample query.

Query: Display for each product the difference between its standard price and the overall average standard price of all products.

```
SELECT ProductStandardPrice – PriceAvg AS Difference
  FROM Product_T, (SELECT AVG(ProductStandardPrice) AS PriceAvg
    FROM Product_T);
```

Result:

```
DIFFERENCE
   –240.63
    –65.63
   –265.63
   –190.63
    359.38
   –115.63
    209.38
    309.38
```

Also, it is easy to confuse the functions COUNT (*) and COUNT. The function COUNT (*), used in the previous query, counts all rows selected by a query, regardless of whether any of the rows contain null values. COUNT tallies only rows that contain values; it ignores all null values.

SUM and AVG can only be used with numeric columns. COUNT, COUNT (*), MIN, and MAX can be used with any data type. Using MIN on a text column, for example, will find the lowest value in the column, the one whose first column is closest to the beginning of the alphabet. SQL implementations interpret the order of the alphabet differently. For example, some systems may start with A–Z, then a–z, then 0–9 and special characters. Others treat upper- and lowercase letters as being equivalent. Still others start with some special characters, then proceed to numbers, letters, and other special characters. Here is the query to ask for the first ProductName in Product_T alphabetically, which was done using the AMERICAN character set in Oracle 11g.

Query: Alphabetically, what is the first product name in the Product table?

```
SELECT MIN (ProductDescription)
  FROM Product_T;
```

It gives the following result, which demonstrates that numbers are sorted before letters in this character set. (Note: The following result is from Oracle. Microsoft SQL Server returns the same result but labels the column (No column name) in SQL Query Analyzer, unless the query specifies a name for the result.)

Result:

```
MIN(PRODUCTDESCRIPTION)
-----------------------
8-Drawer Desk
```

Using Wildcards

The use of the asterisk (*) as a wildcard in a SELECT statement has been previously shown. Wildcards may also be used in the WHERE clause when an exact match is not possible. Here, the keyword LIKE is paired with wildcard characters and usually a string containing the characters that are known to be desired matches. The wildcard character, %, is used to represent any collection of characters. Thus, using LIKE '%Desk' when searching ProductDescription will find all different types of desks carried by Pine Valley Furniture. The underscore (_) is used as a wildcard character to represent exactly one character rather than any collection of characters. Thus, using LIKE '_-drawer' when searching ProductName will find any products with specified drawers, such as 3-, 5-, or 8-drawer dressers.

Using Comparison Operators

With the exception of the very first SQL example in this section, we have used the equality comparison operator in our WHERE clauses. The first example used the greater (less) than operator. The most common comparison operators for SQL implementations are listed in Table 6-3. (Different SQL DBMSs can use different comparison operators.) You are used to thinking about using comparison operators with numeric data, but you can also use them with character data and dates in SQL. The query shown here asks for all orders placed after 10/24/2010.

TABLE 6-3 Comparison Operators in SQL

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
!=	Not equal to

Query: Which orders have been placed since 10/24/2010?

```
SELECT OrderID, OrderDate
  FROM Order_T
    WHERE OrderDate > '24-OCT-2010';
```

Notice that the date is enclosed in single quotes and that the format of the date is different from that shown in Figure 6-3, which was taken from Microsoft Access. The query was run in SQL*Plus. You should check the reference manual for the SQL language you are using to see how dates are to be formatted in queries and for data input.

Result:

ORDERID	ORDERDATE
1007	27-OCT-10
1008	30-OCT-10
1009	05-NOV-10
1010	05-NOV-10

Query: What furniture does Pine Valley carry that isn't made of cherry?

```
SELECT ProductDescription, ProductFinish
  FROM Product_T
    WHERE ProductFinish != 'Cherry';
```

Result:

PRODUCTDESCRIPTION	PRODUCTFINISH
Coffee Table	Natural Ash
Computer Desk	Natural Ash
Entertainment Center	Natural Maple
8-Drawer Desk	White Ash
Dining Table	Natural Ash
Computer Desk	Walnut

Using Null Values

Columns that are defined without the NOT NULL clause may be empty, and this may be a significant fact for an organization. You will recall that a null value means that a column is missing a value; the value is not zero or blank or any special code—there simply is no value. We have already seen that functions may produce different results when null values are present than when a column has a value of zero in all qualified rows. It is not uncommon, then, to first explore whether there are null values before deciding how to write other commands, or it may be that you simply want to see data about table rows where there are missing values. For example, before undertaking a postal mail advertising campaign, you might want to pose the following query.

Query: Display all customers for whom we do not know their postal code.

```
SELECT * FROM Customer_T WHERE CustomerPostalCode IS NULL;
```

Result:

Fortunately, this query returns 0 rows in the result in our sample database, so we can mail advertisements to all our customers because we know their postal codes. The term IS NOT NULL returns results for rows where the qualified column has a non-null value. This allows us to deal with rows that have values in a critical column, ignoring other rows.

Using Boolean Operators

You probably have taken a course or part of a course on finite or discrete mathematics—logic, Venn diagrams, and set theory, oh my! Remember we said that SQL is a set-oriented language, so there are many opportunities to use what you learned in finite math to write complex SQL queries. Some complex questions can be answered by adjusting the WHERE clause further. The Boolean or logical operators AND, OR, and NOT can be used to good purpose:

AND	Joins two or more conditions and returns results only when all conditions are true.
OR	Joins two or more conditions and returns results when any conditions are true.
NOT	Negates an expression.

If multiple Boolean operators are used in an SQL statement, NOT is evaluated first, then AND, then OR. For example, consider the following query.

Query A: List product name, finish, and standard price for all desks and all tables that cost more than $300 in the Product table.

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
  FROM Product_T
    WHERE ProductDescription LIKE '%Desk'
      OR ProductDescription LIKE '%Table'
      AND ProductStandardPrice > 300;
```

Result:

PRODUCTDESCRIPTION	PRODUCTFINISH	PRODUCTSTANDARDPRICE
Computer Desk	Natural Ash	375
Writer's Desk	Cherry	325
8-Drawer Desk	White Ash	750
Dining Table	Natural Ash	800
Computer Desk	Walnut	250

All of the desks are listed, even the computer desk that costs less than $300. Only one table is listed; the less expensive ones that cost less than $300 are not included. With

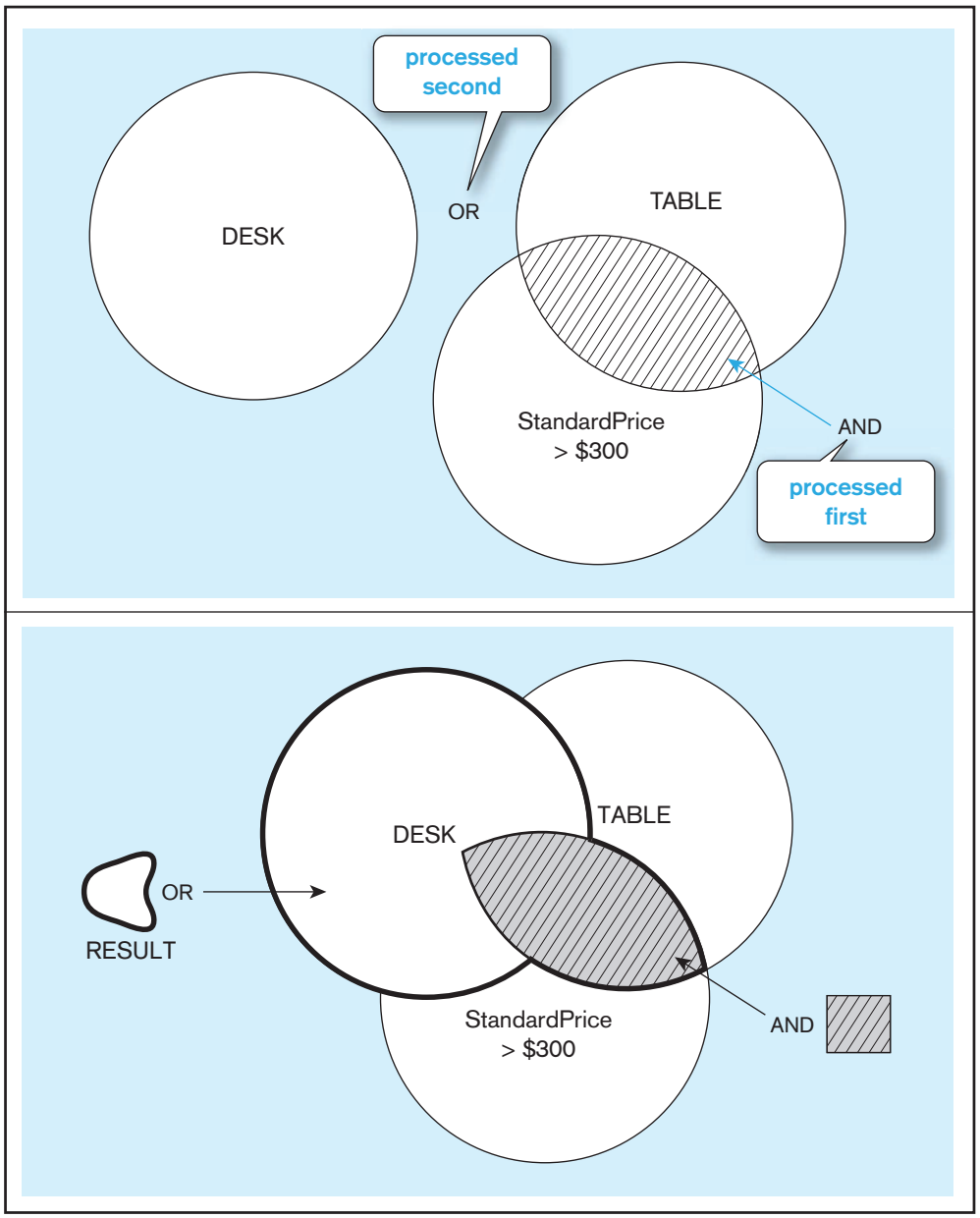


FIGURE 6-8 Boolean query without the use of parentheses
(a) Venn diagram of Query A logic, first process (AND)

(b) Venn diagram of Query A logic, second process (OR)

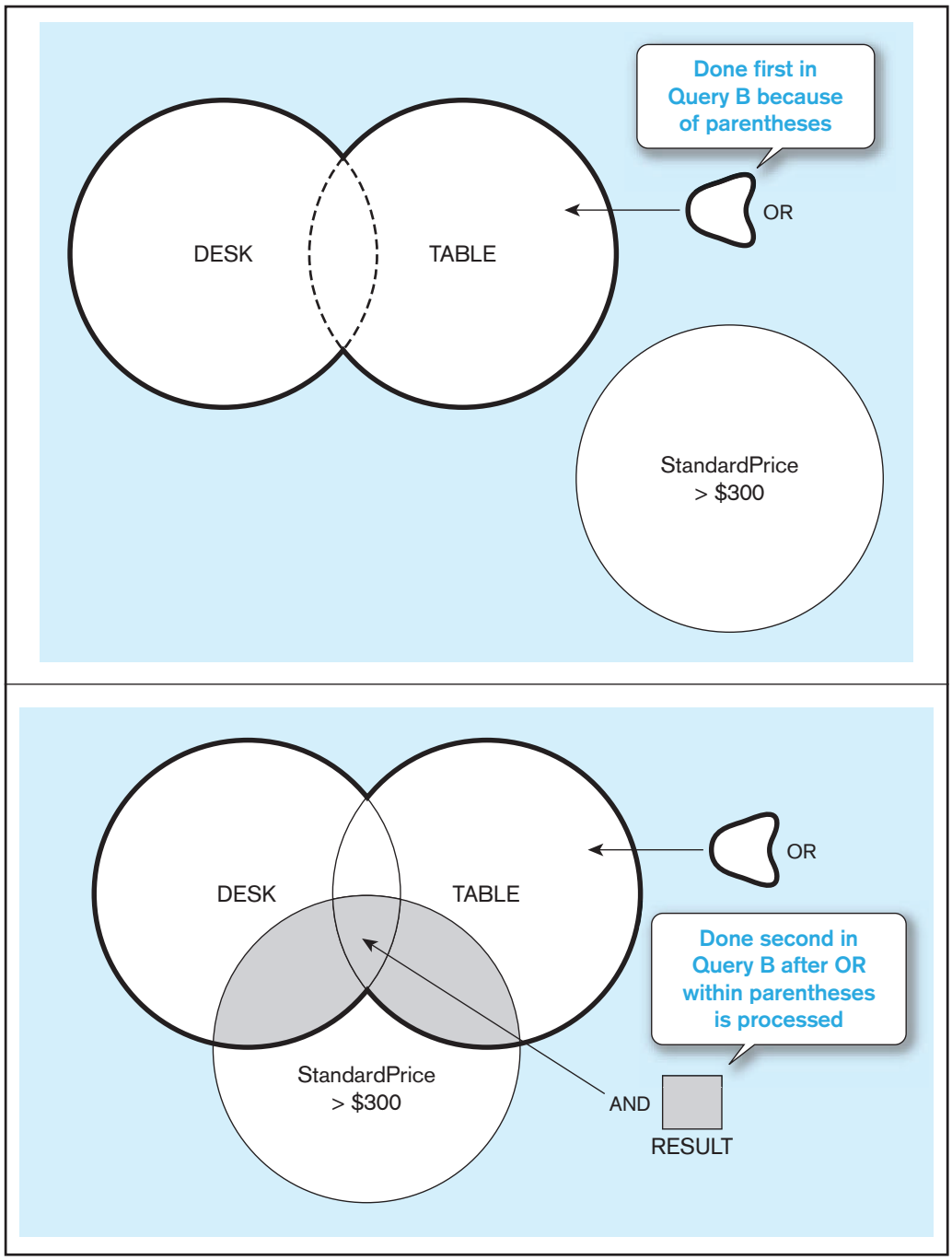
this query, the AND will be processed first, returning all tables with a standard price greater than $300 (Figure 6-8a). Then the OR is processed, returning all desks, regardless of cost, and all tables costing more than $300 (Figure 6-8b). This is the area surrounded by the thick OR line in Figure 6-8b.

If we had wanted to return only desks and tables costing more than $300, we should have put parentheses after the WHERE and before the AND, as shown in Query B below. Figures 6-9a and 6-9b show the difference in processing caused by the judicious use of parentheses in the query. The result is all desks and tables with a standard price of more than $300, indicated by the filled area with horizontal lines. The walnut computer desk has a standard price of $250 and is not included.

Query B: List product name, finish, and unit price for all desks and tables in the PRODUCT table that cost more than $300.

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
  FROM Product_T;
  WHERE (ProductDescription LIKE '%Desk'
    OR ProductDescription LIKE '%Table')
    AND ProductStandardPrice > 300;
```

FIGURE 6-9 **Boolean query with use of parentheses**
(a) Venn diagram of Query B logic, first process (AND)



(b) Venn diagram of Query B, second process (OR)

The results follow. Only products with unit price greater than $300 are included.

Result:

PRODUCTDESCRIPTION	PRODUCTFINISH	PRODUCTSTANDARDPRICE
Computer Desk	Natural Ash	375
Writer's Desk	Cherry	325
8-Drawer Desk	White Ash	750
Dining Table	Natural Ash	800

This example illustrates why SQL is considered a set-oriented, not a record-oriented, language. (C, Java, and Cobol are examples of record-oriented languages because they must process one record, or row, of a table at a time.) To answer this

query, SQL will find the set of rows that are Desk products, and then it will union (i.e., merge) that set with the set of rows that are Table products. Finally, it will intersect (i.e., find common rows) the resultant set from this union with the set of rows that have a standard price above $300. If indexes can be used, the work is done even faster, because SQL will create sets of index entries that satisfy each qualification and do the set manipulation on those index entry sets, each of which takes up less space and can be manipulated much more quickly. You will see in Chapter 7 even more dramatic ways in which the set-oriented nature of SQL works for more complex queries involving multiple tables.

Using Ranges for Qualification

The comparison operators < and > are used to establish a range of values. The keywords BETWEEN and NOT BETWEEN can also be used. For example, to find products with a standard price between $200 and $300, the following query could be used.

Query: Which products in the Product table have a standard price between $200 and $300?

```
SELECT ProductDescription, ProductStandardPrice
  FROM Product_T
    WHERE ProductStandardPrice > 199 AND ProductStandardPrice < 301;
```

Result:

PRODUCTDESCRIPTION	PRODUCTSTANDARDPRICE
Coffee Table	200
Computer Desk	250

The same result will be returned by the following query.

Query: Which products in the PRODUCT table have a standard price between $200 and $300?

```
SELECT ProductDescription, ProductStandardPrice
  FROM Product_T
    WHERE ProductStandardPrice BETWEEN 200 AND 300;
```

Result: Same as previous query.

Adding NOT before BETWEEN in this query will return all the other products in Product_T because their prices are less than $200 or more than $300.

Using Distinct Values

Sometimes when returning rows that don't include the primary key, duplicate rows will be returned. For example, look at this query and the results that it returns.

Query: What order numbers are included in the OrderLine table?

```
SELECT OrderID
  FROM OrderLine_T;
```

Eighteen rows are returned, and many of them are duplicates because many orders were for multiple items.

Result:

ORDERID
1001
1001
1001
1002
1003
1004
1004
1005
1006
1006
1006
1007
1007
1008
1008
1009
1009
1010

18 rows selected.

Do we really need the redundant OrderIDs in this result? If we add the keyword DISTINCT, then only 1 occurrence of each OrderID will be returned, 1 for each of the 10 orders represented in the table.

Query: What are the distinct order numbers included in the OrderLine table?

```
SELECT DISTINCT OrderID
  FROM OrderLine_T;
```

Result:

ORDERID
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010

10 rows selected.

DISTINCT and its counterpart, ALL, can be used only once in a SELECT statement. It comes after SELECT and before any columns or expressions are listed. If a SELECT statement projects more than one column, only rows that are identical for every column

will be eliminated. Thus, if the previous statement also includes OrderedQuantity, 14 rows are returned because there are now only 4 duplicate rows rather than 8. For example, both items ordered on OrderID 1004 were for 2 items, so the second pairing of 1004 and 2 will be eliminated.

Query: What are the unique combinations of order number and order quantity included in the OrderLine table?

```
SELECT DISTINCT OrderID, OrderedQuantity
  FROM OrderLine_T;
```

Result:

ORDERID	ORDEREDQUANTITY
1001	1
1001	2
1002	5
1003	3
1004	2
1005	4
1006	1
1006	2
1007	2
1007	3
1008	3
1009	2
1009	3
1010	10

14 rows selected.

Using IN and NOT IN with Lists

To match a list of values, consider using IN.

Query: List all customers who live in warmer states.

```
SELECT CustomerName, CustomerCity, CustomerState
  FROM Customer_T
    WHERE CustomerState IN ('FL', 'TX', 'CA', 'HI');
```

Result:

CUSTOMERNAME	CUSTOMERCITY	CUSTOMERSTATE
Contemporary Casuals	Gainesville	FL
Value Furniture	Plano	TX
Impressions	Sacramento	CA
California Classics	Santa Clara	CA
M and H Casual Furniture	Clearwater	FL
Seminole Interiors	Seminole	FL
Kaneohe Homes	Kaneohe	HI

7 rows selected.

IN is particularly useful in SQL statements that use subqueries, which will be covered in Chapter 7. The use of IN is also very consistent with the set nature of SQL. Very simply, the list (set of values) inside the parentheses after IN can be literals, as illustrated here, or can be a SELECT statement with a single result column, the result of which will be plugged in as the set of values for comparison. In fact, some SQL programmers always use IN, even when the set in parentheses after IN includes only one item. Similarly, any "table" of the FROM clause can be itself a derived table defined by including a SELECT statement in parentheses in the FROM clause (as we saw earlier, with the query about the difference between the standard price of each product and the average standard price of all products). The ability to include a SELECT statement anyplace within SQL where a set is involved is a very powerful and useful feature of SQL, and, of course, totally consistent with SQL being a set-oriented language, as illustrated in Figures 6-8 and 6-9.

Sorting Results: The ORDER BY Clause

Looking at the preceding results, it may seem that it would make more sense to list the California customers, followed by the Floridians, Hawaiians, and Texans. That brings us to the other three basic parts of the SQL statement:

ORDER BY	Sorts the final results rows in ascending or descending order.
GROUP BY	Groups rows in an intermediate results table where the values in those rows are the same for one or more columns.
HAVING	Can only be used following a GROUP BY and acts as a secondary WHERE clause, returning only those groups that meet a specified condition.

So, we can order the customers by adding an ORDER BY clause.

Query: List customer, city, and state for all customers in the Customer table whose address is Florida, Texas, California, or Hawaii. List the customers alphabetically by state and alphabetically by customer within each state.

```
SELECT CustomerName, CustomerCity, CustomerState
  FROM Customer_T
    WHERE CustomerState IN ('FL', 'TX', 'CA', 'HI')
      ORDER BY CustomerState, CustomerName;
```

Now the results are easier to read.

Result:

CUSTOMERNAME	CUSTOMERCITY	CUSTOMERSTATE
California Classics	Santa Clara	CA
Impressions	Sacramento	CA
Contemporary Casuals	Gainesville	FL
M and H Casual Furniture	Clearwater	FL
Seminole Interiors	Seminole	FL
Kaneohe Homes	Kaneohe	HI
Value Furniture	Plano	TX
7 rows selected.		

Notice that all customers from each state are listed together, and within each state, customer names are alphabetized. The sorting order is determined by the order in which the columns are listed in the ORDER BY clause; in this case, states were alphabetized first, then customer names. If sorting from high to low, use DESC as a keyword, placed after the column used to sort. Instead of typing the column names in the ORDER BY clause, you can use their column positions in the select list; for example, in the preceding query, we could have written the clause as

```
ORDER BY 3, 1;
```

For cases in which there are many rows in the result table but you need to see only a few of them, many SQL systems (including MySQL) support a LIMIT clause, such as the following, which would show only the first five rows of the result:

```
ORDER BY 3, 1 LIMIT 5;
```

The following would show five rows after skipping the first 30 rows:

```
ORDER BY 3, 1 LIMIT 30, 5;
```

How are NULLs sorted? Null values may be placed first or last, before or after columns that have values. Where the NULLs will be placed will depend upon the SQL implementation.

Categorizing Results: The GROUP BY Clause

GROUP BY is particularly useful when paired with aggregate functions, such as SUM or COUNT. GROUP BY divides a table into subsets (by groups); then an aggregate function can be used to provide summary information for that group. The single value returned by the previous aggregate function examples is called a **scalar aggregate**. When aggregate functions are used in a GROUP BY clause and several values are returned, they are called **vector aggregates**.

Scalar aggregate
A single value returned from an SQL query that includes an aggregate function.

Vector aggregate
Multiple values returned from an SQL query that includes an aggregate function.

Query: Count the number of customers with addresses in each state to which we ship.

```
SELECT CustomerState, COUNT (CustomerState)
  FROM Customer_T
    GROUP BY CustomerState;
```

Result:

CUSTOMERSTATE	COUNT(CUSTOMERSTATE)
CA	2
CO	1
FL	3
HI	1
MI	1
NJ	2
NY	1
PA	1
TX	1
UT	1
WA	1

11 rows selected.

It is also possible to nest groups within groups; the same logic is used as when sorting multiple items.

Query: Count the number of customers with addresses in each city to which we ship. List the cities by state.

```
SELECT CustomerState, CustomerCity, COUNT (CustomerCity)
  FROM Customer_T
    GROUP BY CustomerState, CustomerCity;
```

Although the GROUP BY clause seems straightforward, it can produce unexpected results if the logic of the clause is forgotten (and this is a common "gotcha" for novice SQL coders). When a GROUP BY is included, the columns allowed to be specified in the SELECT clause are limited. Only a column with a single value for each group can be included. In the previous query, each group is identified by the combination of a city and its state. The SELECT statement includes both the city and state columns. This works because each combination of city and state is one COUNT value. But if the SELECT clause of the first query in this section had also included city, that statement would fail because the GROUP BY is only by state. Because a state can have more than one city, the requirement that each value in the SELECT clause have only one value in the GROUP BY group is not met, and SQL will not be able to present the city information so that it makes sense. *If you write queries using the following rule, your queries will work*: Each column referenced in the SELECT statement must be referenced in the GROUP BY clause, unless the column is an argument for an aggregate function included in the SELECT clause.

Qualifying Results by Categories: The HAVING Clause

The HAVING clause acts like a WHERE clause, but it identifies groups, rather than rows, that meet a criterion. Therefore, you will usually see a HAVING clause following a GROUP BY clause.

Query: Find only states with more than one customer.

```
SELECT CustomerState, COUNT (CustomerState)
  FROM Customer_T
    GROUP BY CustomerState
    HAVING COUNT (CustomerState) > 1;
```

This query returns a result that has removed all states (groups) with one customer. Remember that using WHERE here would not work because WHERE doesn't allow aggregates; further, WHERE qualifies a set of rows, whereas HAVING qualifies a set of groups. As with WHERE, the HAVING qualification can be compared to the result of a SELECT statement, which computes the value for comparison (i.e., a set with only one value is still a set).

Result:

CUSTOMERSTATE	COUNT(CUSTOMERSTATE)
CA	2
FL	3
NJ	2

To include more than one condition in the HAVING clause, use AND, OR, and NOT just as in the WHERE clause. In summary, here is one last command that includes all six clauses; remember that they must be used in this order.

Query: List, in alphabetical order, the product finish and the average standard price for each finish for selected finishes having an average standard price less than 750.

```
SELECT ProductFinish, AVG (ProductStandardPrice)
  FROM Product_T
    WHERE ProductFinish IN ('Cherry', 'Natural Ash', 'Natural Maple',
    'White Ash')
      GROUP BY ProductFinish
        HAVING AVG (ProductStandardPrice) < 750
          ORDER BY ProductFinish;
```

Result:

PRODUCTFINISH	AVG(PRODUCTSTANDARDPRICE)
Cherry	250
Natural Ash	458.333333
Natural Maple	650

Figure 6-10 shows the order in which SQL processes the clauses of a statement. Arrows indicate the paths that may be followed. Remember, only the SELECT and FROM clauses are mandatory. Notice that the processing order is different from the order of the syntax used to create the statement. As each clause is processed, an intermediate results table is produced that will be used for the next clause. Users do not see the intermediate results tables; they see only the final results. A query can be debugged by remembering the order shown in Figure 6-10. Take out the optional clauses and then add them back in one at a time in the order in which they will be processed. In this way, intermediate results can be seen and problems often can be spotted.

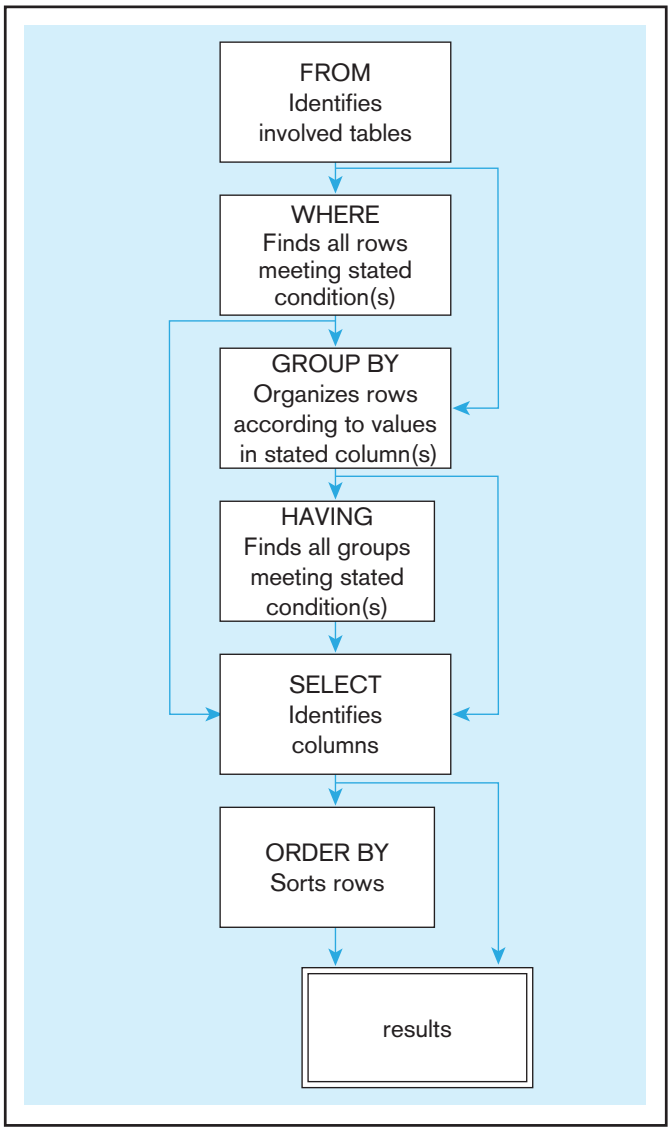


FIGURE 6-10 **SQL statement processing order (adapted from van der Lans, 2006, p. 100)**

Using and Defining Views

Base table
A table in the relational data model containing the inserted raw data. Base tables correspond to the relations that are identified in the database's conceptual schema.

Virtual table
A table constructed automatically as needed by a DBMS. Virtual tables are not maintained as real data.

Dynamic view
A virtual table that is created dynamically upon request by a user. A dynamic view is not a temporary table. Rather, its definition is stored in the system catalog, and the contents of the view are materialized as a result of an SQL query that uses the view. It differs from a materialized view, which may be stored on a disk and refreshed at intervals or when used, depending on the RDBMS.

Materialized view
Copies or replicas of data, based on SQL queries created in the same manner as dynamic views. However, a materialized view exists as a table and thus care must be taken to keep it synchronized with its associated base tables.

The SQL syntax shown in Figure 6-6 demonstrates the creation of four **base tables** in a database schema using Oracle 11g SQL. These tables, which are used to store data physically in the database, correspond to relations in the logical database design. By using SQL queries with any RDBMS, it is possible to create **virtual tables**, or **dynamic views**, whose contents materialize when referenced. These views may often be manipulated in the same way as a base table can be manipulated, through SQL SELECT queries. **Materialized views**, which are stored physically on a disk and refreshed at appropriate intervals or events, may also be used.

The often-stated purpose of a view is to simplify query commands, but a view may also improve data security and significantly enhance programming consistency and productivity for a database. To highlight the convenience of using a view, consider Pine Valley's invoice processing. Construction of the company's invoice requires access to the four tables from the Pine Valley database of Figure 6-3: Customer_T, Order_T, OrderLine_T, and Product_T. A novice database user may make mistakes or be unproductive in properly formulating queries involving so many tables. A view allows us to predefine this association into a single virtual table as part of the database. With this view, a user who wants only customer invoice data does not have to reconstruct the joining of tables to produce the report or any subset of it. Table 6-4 summarizes the pros and cons of using views.

A view, Invoice_V, is defined by specifying an SQL query (SELECT . . . FROM . . . WHERE) that has the view as its result. If you decide to try this query as is, without selecting additional attributes, remove the comma after OrderedQuantity. The example assumes you will elect to include additional attributes in the query.

Query: What are the data elements necessary to create an invoice for a customer? Save this query as a view named Invoice_V.

```
CREATE VIEW Invoice_V AS
  SELECT  Customer_T.CustomerID,  CustomerAddress,  Order_T.OrderID,
  Product_T.ProductID,ProductStandardPrice,
  OrderedQuantity, and other columns as required
    FROM Customer_T, Order_T, OrderLine_T, Product_T
      WHERE Customer_T.CustomerID = Order_T.CustomerID
        AND Order_T.OrderID = OrderLine_T.OrderD
        AND Product_T.ProductID = OrderLine_T.ProductID;
```

The SELECT clause specifies, or projects, what data elements (columns) are to be included in the view table. The FROM clause lists the tables and views involved in the view development. The WHERE clause specifies the names of the common columns used to join Customer_T to Order_T to OrderLine_T to Product_T. (You'll learn about joining in Chapter 7, but for now remember the foreign keys that were defined to reference other tables; these are the columns used for joining.) Because a view is a table, and one of the relational properties of tables is that the order of rows is immaterial, the rows

TABLE 6-4 Pros and Cons of Using Dynamic Views

Positive Aspects	Negative Aspects
Simplify query commands	Use processing time re-creating the view each time it is referenced
Help provide data security and confidentiality	May or may not be directly updateable
Improve programmer productivity	
Contain most current base table data	
Use little storage space	
Provide a customized view for a user	
Establish physical data independence	

in a view may not be sorted. But queries that refer to this view may display their results in any desired sequence.

We can see the power of such a view when building a query to generate an invoice for order number 1004. Rather than specify the joining of four tables, we can have the query include all relevant data elements from the view table, Invoice_V.

Query: What are the data elements necessary to create an invoice for order number 1004?

```
SELECT CustomerID, CustomerAddress, ProductID,
  OrderedQuantity, and other columns as required
    FROM Invoice_V
      WHERE OrderID = 1004;
```

A dynamic view is a virtual table; it is constructed automatically, as needed, by the DBMS and is not maintained as persistent data. Any SQL SELECT statement may be used to create a view. The persistent data are stored in base tables, those that have been defined by CREATE TABLE commands. A dynamic view always contains the most current derived values and is thus superior in terms of data currency to constructing a temporary real table from several base tables. Also, in comparison to a temporary real table, a view consumes very little storage space. A view is costly, however, because its contents must be calculated each time they are requested (that is, each time the view is used in an SQL statement). Materialized views are now available and address this drawback.

A view may join together multiple tables or views and may contain derived (or virtual) columns. For example, if a user of the Pine Valley Furniture database only wants to know the total value of orders placed for each furniture product, a view for this can be created from Invoice_V. The following example in SQL*Plus illustrates how this is done with Oracle, although this can be done with any RDBMS that supports views.

Query: What is the total value of orders placed for each furniture product?

```
CREATE VIEW OrderTotals_V AS
  SELECT ProductID Product, SUM (ProductStandardPrice*OrderedQuantity)
  Total
    FROM Invoice_V
    GROUP BY ProductID;
```

We can assign a different name (an alias) to a view column rather than use the associated base table or expression column name. Here, Product is a renaming of ProductID, local to only this view. Total is the column name given the expression for total sales of each product. (Total may not be a legal alias with some relational DBMSs because it might be a reserved word for a proprietary function of the DBMS; you always have to be careful when defining columns and aliases not to use a reserved word.) The expression can now be referenced via this view in subsequent queries as if it were a column rather than a derived expression. Defining views based on other views can cause problems. For example, if we redefine Invoice_V so that StandardPrice is not included, then OrderTotals_V will no longer work because it will not be able to locate standard unit prices.

Views can also help to establish security. Tables and columns that are not included will not be obvious to the user of the view. Restricting access to a view with GRANT and REVOKE statements adds another layer of security. For example, granting some users access rights to aggregated data, such as averages, in a view but denying them access to detailed base table data will not allow them to display the base table data. SQL security commands are explained further in Chapter 11.

Privacy and confidentiality of data can be achieved by creating views that restrict users to working with only the data they need to perform their assigned duties. If a clerical worker needs to work with employees' addresses but should not be able to access their compensation rates, they may be given access to a view that does not contain compensation information.

Some people advocate the creation of a view for every single base table, even if that view is identical to the base table. They suggest this approach because views

can contribute to greater programming productivity as databases evolve. Consider a situation in which 50 programs all use the Customer_T table. Suppose that the Pine Valley Furniture Company database evolves to support new functions that require the Customer_T table to be renormalized into two tables. If these 50 programs refer directly to the Customer_T base table, they will all have to be modified to refer to one of the two new tables or to joined tables. But if these programs all use the view on this base table, then only the view has to be re-created, saving considerable reprogramming effort. However, dynamic views require considerable run-time computer processing because the virtual table of a view is re-created each time the view is referenced. Therefore, referencing a base table through a view rather than directly can add considerable time to query processing. This additional operational cost must be balanced against the potential reprogramming savings from using a view.

It can be possible to update base table data via update commands (INSERT, DELETE, and UPDATE) against a view as long as it is unambiguous what base table data must change. For example, if the view contains a column created by aggregating base table data, then it would be ambiguous how to change the base table values if an attempt were made to update the aggregate value. If the view definition includes the WITH CHECK OPTION clause, attempts to insert data through the view will be rejected when the data values do not meet the specifications of WITH CHECK OPTION. Specifically, when the CREATE VIEW statement contains any of the following situations, that view may not be used to update the data:

1. The SELECT clause includes the keyword DISTINCT.
2. The SELECT clause contains expressions, including derived columns, aggregates, statistical functions, and so on.
3. The FROM clause, a subquery, or a UNION clause references more than one table.
4. The FROM clause or a subquery references another view that is not updateable.
5. The CREATE VIEW command contains a GROUP BY or HAVING clause.

It could happen that an update to an instance would result in the instance disappearing from the view. Let's create a view named ExpensiveStuff_V, which lists all furniture products that have a StandardPrice over $300. That view will include ProductID 5, a writer's desk, which has a unit price of $325. If we update data using Expensive_Stuff_V and reduce the unit price of the writer's desk to $295, then the writer's desk will no longer appear in the ExpensiveStuff_V virtual table because its unit price is now less than $300. In Oracle, if you want to track all merchandise with an original price over $300, include a WITH CHECK OPTION clause after the SELECT clause in the CREATE VIEW command. WITH CHECK OPTION will cause UPDATE or INSERT statements on that view to be rejected when those statements would cause updated or inserted rows to be removed from the view. This option can be used only with updateable views.

Here is the CREATE VIEW statement for ExpensiveStuff_V.

Query: List all furniture products that have ever had a standard price over $300.

```
CREATE VIEW ExpensiveStuff_V
  AS
    SELECT ProductID, ProductDescription, ProductStandardPrice
      FROM Product_T
        WHERE ProductStandardPrice > 300
        WITH CHECK OPTION;
```

When attempting to update the unit price of the writer's desk to $295 using the following Oracle SQL*Plus syntax:

```
UPDATE ExpensiveStuff_V
SET ProductStandardPrice = 295
  WHERE ProductID = 5;
```

Oracle gives the following error message:

```
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

A price increase on the writer's desk to $350 will take effect with no error message because the view is updateable and the conditions specified in the view are not violated.

Information about views will be stored in the systems tables of the DBMS. In Oracle 11g, for example, the text of all views is stored in DBA_VIEWS. Users with system privileges can find this information.

Query: List some information that is available about the view named EXPENSIVESTUFF_V. (Note that EXPENSIVESTUFF_V is stored in uppercase and must be entered in uppercase in order to execute correctly.)

```
SELECT OWNER,VIEW_NAME,TEXT_LENGTH FROM DBA_VIEWS
  WHERE VIEW_NAME = 'EXPENSIVESTUFF_V';
```

Result:

OWNER	VIEW_NAME	TEXT_LENGTH
MPRESCOTT	EXPENSIVESTUFF_V	110

MATERIALIZED VIEWS Like dynamic views, materialized views can be constructed in different ways for various purposes. Tables may be replicated in whole or in part and refreshed on a predetermined time interval or triggered when the table needs to be accessed. Materialized views can be based on queries from one or more tables. It is possible to create summary tables based on aggregations of data. Copies of remote data that use distributed data may be stored locally as materialized views. Maintenance overhead will be incurred to keep the local view synchronized with the remote base tables or data warehouse, but the use of materialized views may improve the performance of distributed queries, especially if the data in the materialized view are relatively static and do not have to be refreshed very often.

Summary

This chapter has introduced the SQL language for relational database definition (DDL), manipulation (DML), and control (DCL) languages, commonly used to define and query relational database management systems (RDBMSs). This standard has been criticized as having many flaws. In reaction to these criticisms and to increase the power of the language, extensions are constantly under review by the ANSI X3H2 committee and International Committee for Information Technology Standards (INCITS). The current generally implemented standard is SQL:1999, but SQL:2008 is under final draft review.

The establishment of SQL standards and conformance certification tests has contributed to relational systems being the dominant form of new database development. Benefits of the SQL standards include reduced training costs, improved productivity, application portability and longevity, reduced dependence on single vendors, and improved cross-system communication.

The SQL environment includes an instance of an SQL DBMS along with accessible databases and associated users and programs. Each database is included in a catalog and has a schema that describes the database objects. Information contained in the catalog is maintained by the DBMS itself rather than by the users of the DBMS.

The SQL DDL commands are used to define a database, including its creation and the creation of its tables, indexes, and views. Referential integrity is also established through DDL commands. The SQL DML commands are used to load, update, and query the database through use of the SELECT command. DCL commands are used to establish user access to the database.

SQL commands may directly affect the base tables, which contain the raw data, or they may affect a database view that has been created. Changes and updates made to views may or may not be passed on to the base tables. The basic syntax of an SQL SELECT statement contains the following keywords: SELECT, FROM, WHERE, ORDER BY, GROUP BY, and HAVING. SELECT determines which attributes will be displayed in the query results table. FROM determines which tables or views will be used in the query. WHERE sets the criteria of the query, including any joins of multiple tables that are necessary. ORDER BY determines the order in which the results will be displayed. GROUP BY is used to categorize results and may return either scalar aggregates or vector aggregates. HAVING qualifies results by categories.

Understanding the basic SQL syntax presented in this chapter should enable the reader to start using SQL effectively and to build a deeper understanding of the possibilities for more complex querying with continued practice. Advanced SQL topics are covered in Chapter 7.

Chapter Review

Key Terms

Base table *278*
Catalog *247*
Data control language (DCL) *248*
Data definition language (DDL) *248*
Data manipulation language (DML) *248*
Dynamic view *278*
Materialized view *278*
Relational DBMS (RDBMS) *247*
Scalar aggregate *275*
Schema *247*
Vector aggregate *275*
Virtual table *278*

Review Questions

1. Define each of the following terms:
   a. base table
   b. data definition language
   c. data manipulation language
   d. dynamic view
   e. materialized view
   f. referential integrity constraint
   g. relational DBMS (RDBMS)
   h. schema
   i. virtual table

2. Match the following terms to the appropriate definitions:

____ view	a. list of values
____ referential integrity constraint	b. description of a database
____ dynamic view	c. view materialized as a result of a SQL query that uses the view
____ materialized view	d. logical table
____ SQL:200n	e. missing or nonexistent value
____ null value	f. descriptions of database objects of a database
____ scalar aggregate	g. programming language in which SQL commands are embedded
____ vector aggregate	h. established in relational data models by use of foreign keys
____ catalog	i. view that exists as a table
____ schema	j. currently proposed standard relational query and definition language
____ host language	k. single value

3. Contrast the following terms:
   a. base table; view
   b. dynamic view; materialized view
   c. catalog; schema

4. What are SQL-92, SQL:1999, and SQL:200n? Briefly describe how SQL:200n differs from SQL:1999.

5. Describe a relational DBMS (RDBMS), its underlying data model, its data storage structures, and how data relationships are established.

6. List six potential benefits of achieving an SQL standard that is widely accepted.

7. Describe the components and structure of a typical SQL environment.

8. Distinguish among data definition commands, data manipulation commands, and data control commands.

9. Explain how referential integrity is established in databases that are SQL:1999 compliant. Explain how the ON UPDATE RESTRICT, ON UPDATE CASCADE, and ON UPDATE SET NULL clauses differ from one another. What happens if the ON DELETE CASCADE clause is set?

10. Explain some possible purposes of creating a view using SQL. In particular, explain how a view can be used to reinforce data security.

11. Explain why it is necessary to limit the kinds of updates performed on data when referencing data through a view.

12. Describe a set of circumstances for which using a view can save reprogramming effort.

13. Drawing on material covered in prior chapters, explain the factors to be considered in deciding whether to create a key index for a table in SQL.

14. Explain and provide at least one example of how to qualify the ownership of a table in SQL. What has to occur for one user to be allowed to use a table in a database owned by another user?

15. How is the order in which attributes appear in a result table changed? How are the column heading labels in a result table changed?

16. What is the difference between COUNT, COUNT DISTINCT, and COUNT(*) in SQL? When will these three commands generate the same and different results?

17. What is the evaluation order for the Boolean operators (AND, OR, NOT) in an SQL command? How can one be sure that the operators will work in the desired order rather than in this prescribed order?

18. If an SQL statement includes a GROUP BY clause, the attributes that can be requested in the SELECT statement will be limited. Explain that limitation.

19. Describe a situation in which you would need to write a query using the HAVING clause.

20. In what clause of a SELECT statement is an IN operator used? What follows the IN operator? What other SQL operator can sometimes be used to perform the same operation as the IN operator? Under what circumstances can this other operator be used?

21. Explain why SQL is called a set-oriented language.

22. When would the use of the LIKE keyword with the CREATE TABLE command be helpful?

23. What is an identity column? Explain the benefits of using the identity column capability in SQL.

24. SQL:200n has a new keyword, MERGE. Explain how using this keyword allows one to accomplish updating and merging data into a table using one command rather than two.

25. In what order are the clauses of an SQL statement processed?

26. Within which clauses of an SQL statement can a derived table be defined?

27. In an ORDER BY clause, what are the two ways to refer to the columns to be used for sorting the results of the query?

28. Explain the purpose of the CHECK clause within a CREATE TABLE SQL command. Explain the purpose of the WITH CHECK OPTION in a CREATE VIEW SQL command.
29. What can be changed about a table definition, using the ALTER SQL command? Can you identify anything about a table definition that cannot be changed using the ALTER SQL command?
30. Is it possible to use both a WHERE clause and a HAVING clause in the same SQL SELECT statement? If so, what are the different purposes of these two clauses?

Problems and Exercises

Problems and Exercises 1 through 9 are based on the class scheduling 3NF relations along with some sample data shown in Figure 6-11. Not shown in this figure are data for an ASSIGNMENT relation, which represents a many-to-many relationship between faculty and sections.

1. Write a database description for each of the relations shown, using SQL DDL (shorten, abbreviate, or change any data names, as needed for your SQL version). Assume the following attribute data types:

StudentID (integer, primary key)
StudentName (25 characters)
FacultyID (integer, primary key)
FacultyName (25 characters)
CourseID (8 characters, primary key)

STUDENT (StudentID, StudentName)

StudentID	StudentName
38214	Letersky
54907	Altvater
66324	Aiken
70542	Marra
...	

QUALIFIED (FacultyID, CourseID, DateQualified)

FacultyID	CourseID	DateQualified
2143	ISM 3112	9/1988
2143	ISM 3113	9/1988
3467	ISM 4212	9/1995
3467	ISM 4930	9/1996
4756	ISM 3113	9/1991
4756	ISM 3112	9/1991
...		

FACULTY (FacultyID, FacultyName)

FacultyID	FacultyName
2143	Birkin
3467	Berndt
4756	Collins
...	

SECTION (SectionNo, Semester, CourseID)

SectionNo	Semester	CourseID
2712	I-2008	ISM 3113
2713	I-2008	ISM 3113
2714	I-2008	ISM 4212
2715	I-2008	ISM 4930
...		

COURSE (CourseID, CourseName)

CourseID	CourseName
ISM 3113	Syst Analysis
ISM 3112	Syst Design
ISM 4212	Database
ISM 4930	Networking
...	

REGISTRATION (StudentID, SectionNo, Semester)

StudentID	SectionNo	Semester
38214	2714	I-2008
54907	2714	I-2008
54907	2715	I-2008
66324	2713	I-2008
...		

FIGURE 6-11 Class scheduling relations (missing ASSIGNMENT)

CourseName (15 characters)
DateQualified (date)
SectionNo (integer, primary key)
Semester (7 characters)

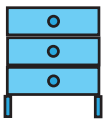
2. Use SQL to define the following view:

StudentID	StudentName
38214	Letersky
54907	Altvater
54907	Altvater
66324	Aiken

3. Because of referential integrity, before any row can be entered into the SECTION table, the CourseID to be entered must already exist in the COURSE table. Write an SQL assertion that will enforce this constraint.
4. Write SQL data definition commands for each of the following queries:
   a. How would you add an attribute, Class, to the Student table?
   b. How would you remove the Registration table?
   c. How would you change the FacultyName field from 25 characters to 40 characters?
5. Write SQL commands for the following:
   a. Create two different forms of the INSERT command to add a student with a student ID of 65798 and last name Lopez to the Student table.
   b. Now write a command that will remove Lopez from the Student table.
   c. Create an SQL command that will modify the name of course ISM 4212 from Database to Introduction to Relational Databases.
6. Write SQL queries to answer the following questions:
   a. Which students have an ID number that is less than 50000?
   b. What is the name of the faculty member whose ID is 4756?
   c. What is the smallest section number used in the first semester of 2008?
7. Write SQL queries to answer the following questions:
   a. How many students are enrolled in Section 2714 in the first semester of 2008?
   b. Which faculty members have qualified to teach a course since 1993? List the faculty ID, course, and date of qualification.
8. Write SQL queries to answer the following questions:
   a. Which students are enrolled in Database and Networking? (Hint: Use SectionNo for each class so you can determine the answer from the Registration table by itself.)
   b. Which instructors cannot teach both Syst Analysis and Syst Design?
9. Write SQL queries to answer the following questions:
   a. What are the courses included in the Section table? List each course only once.
   b. List all students in alphabetical order by StudentName.
   c. List the students who are enrolled in each course in Semester I, 2008. Group the students by the sections in which they are enrolled.
   d. List the courses available. Group them by course prefix. (ISM is the only prefix shown, but there are many others throughout the university.)

Problems and Exercises 10 through 15 are based on the relations shown in Figure 6-12. The database tracks an adult literacy program. Tutors complete a certification class offered by the agency. Students complete an assessment interview that results in a report for the tutor and a recorded Read score. When matched with a student, a tutor meets with the student for one to four hours per week. Some students work with the same tutor for years, some for less than a month. Other students change tutors if their learning style does not match the tutor's tutoring style. Many tutors are retired and are available to tutor only part of the year. Tutor status is recorded as Active, Temp Stop, or Dropped.

10. How many tutors have a status of Temp Stop? Which tutors are active?
11. List the tutors who took the certification class in January.
12. How many students were matched with someone in the first five months of the year?
13. Which student has the highest Read score?
14. How long had each student studied in the adult literacy program?
15. What is the average length of time a student stayed (or has stayed) in the program?



Problems and Exercises 16 through 43 are based on the entire ("big" version) Pine Valley Furniture Company database. Note: Depending on what DBMS you are using, some field names may have changed to avoid using reserved words for the DBMS. When you first use the DBMS, check the table definitions to see what the exact field names are for the DBMS you are using. See the Preface and inside covers of this book for instructions on where to find this database on **www.teradatastudentnetwork.com.**

16. Modify the Product_T table by adding an attribute QtyOnHand that can be used to track the finished goods inventory. The field should be an integer field of five characters and should accept only positive numbers.
17. Enter sample data of your own choosing into QtyOnHand in the Product_T table. Test the modification you made in Problem and Exercise 16 by attempting to update a product by changing the inventory to 10,000 units. Test it again by changing the inventory for the product to –10 units. If you do not receive error messages and are successful in making these changes, then you did not establish appropriate constraints in Problem and Exercise 16.
18. Add an order to the Order_T table and include a sample value for every attribute.
    a. First, look at the data in the Customer_T table and enter an order from any one of those customers.
    b. Enter an order from a new customer. Unless you have also inserted information about the new customer in the Customer_T table, your entry of the order data should be rejected. Referential integrity constraints should prevent you from entering an order if there is no information about the customer.
19. Use the Pine Valley database to answer the following questions:
    a. How many work centers does Pine Valley have?
    b. Where are they located?
20. List the employees whose last names begin with an *L*.
21. Which employees were hired during 1999?
22. List the customers who live in California or Washington. Order them by zip code, from high to low.
23. List all raw materials that are made of cherry and that have dimensions (thickness and width) of 12 by 12.

FIGURE 6-12 **Adult literacy program (for Problems and Exercises 10 through 15)**

TUTOR (TutorID, CertDate, Status)

TutorID	CertDate	Status
100	1/05/2008	Active
101	1/05/2008	Temp Stop
102	1/05/2008	Dropped
103	5/22/2008	Active
104	5/22/2008	Active
105	5/22/2008	Temp Stop
106	5/22/2008	Active

STUDENT (StudentID, Read)

StudentID	Read
3000	2.3
3001	5.6
3002	1.3
3003	3.3
3004	2.7
3005	4.8
3006	7.8
3007	1.5

MATCH HISTORY (MatchID, TutorID, StudentID, StartDate, EndDate)

MatchID	TutorID	StudentID	StartDate	EndDate
1	100	3000	1/10/2008	
2	101	3001	1/15/2008	5/15/2008
3	102	3002	2/10/2008	3/01/2008
4	106	3003	5/28/2008	
5	103	3004	6/01/2008	6/15/2008
6	104	3005	6/01/2008	6/28/2008
7	104	3006	6/01/2008	

24. List the MaterialID, MaterialName, Material, MaterialStandardPrice, and Thickness for all raw materials made of cherry, pine, or walnut. Order the listing by Material, StandardPrice, and Thickness.

25. Display the product line ID and the average standard price for all products in each product line.

26. For every product that has been ordered, display the product ID and the total quantity ordered (label this result TotalOrdered). List the most popular product first and the least popular last.

27. For each customer, list the CustomerID and total number of orders placed.

28. For each salesperson, display a list of CustomerIDs.
29. Display the product ID and the number of orders placed for each product. Show the results in decreasing order by the number of times the product has been ordered and label this result column NumOrders.
30. For each customer, list the CustomerID and the total number of orders placed in 2010.
31. For each salesperson, list the total number of orders.
32. For each customer who had more than two orders, list the CustomerID and the total number of orders placed.
33. List all sales territories (TerritoryID) that have more than one salesperson.
34. Which product is ordered most frequently?
35. Display the territory ID and the number of salespersons in the territory for all territories that have more than one salesperson. Label the number of salespersons NumSalesPersons.
36. Display the SalesPersonID and a count of the number of orders for that salesperson for all salespersons except salespersons 3, 5, and 9. Write this query with as few clauses or components as possible, using the capabilities of SQL as much as possible.
37. For each salesperson, list the total number of orders by month for the year 2010. (Hint: If you are using Access, use the Month function. If you are using Oracle, convert the date to a string, using the TO_CHAR function, with the format string 'Mon' [i.e., TO_CHAR(order_date,'MON')]. If you are using another DBMS, you will need to investigate how to deal with months for this query.)
38. List MaterialName, Material, and Width for raw materials that are *not* cherry or oak and whose width is greater than 10 inches.
39. List ProductID, ProductDescription, ProductFinish, and ProductStandardPrice for oak products with a ProductStandardPrice greater than $400 or cherry products with a StandardPrice less than $300
40. For each order, list the order ID, customer ID, order date, and most recent date among all orders.
41. For each customer, list the customer ID, the number of orders from that customer, and the ratio of the number of orders from that customer to the total number of orders from all customers combined. (This ratio, of course, is the percentage of all orders placed by each customer.)
42. For products 1, 2, and 7, list in one row and three respective columns that product's total unit sales; label the three columns Prod1, Prod2, and Prod7.
43. Not all versions of this database include referential integrity constraints for all foreign keys. Use whatever commands are available for the RDBMS you are using, investigate if any referential integrity constraints are missing. Write any missing constraints and, if possible, add them to the associated table definitions.
44. Tyler Richardson set up a house alarm system when he moved to his new home in Seattle. For security purposes, he has all of his mail, including his alarm system bill, mailed to his local UPS store. Although the alarm system is activated and the company is aware of its physical address, Richardson receives repeated offers mailed to his physical address, imploring him to protect his house with the system he currently uses. What do you think the problem might be with that company's database(s)?

Field Exercises

1. Arrange an interview with a database administrator in an organization in your area. When you interview the database administrator, familiarize yourself with one application that is actively used in the organization. Focus your interview questions on determining end users' involvement with the application and understanding the extent to which end users must be familiar with SQL. For example, if end users are using SQL, what training do they receive? Do they use an interactive form of SQL for their work, or do they use embedded SQL? How have the required skills of the end users changed over the past few years, as the database user interfaces have changed?
2. Arrange an interview with a database administrator in your area. Focus the interview on understanding the environment within which SQL is used in the organization. Inquire about the version of SQL that is used and determine whether the same version is used at all locations. If different versions are used, explore any difficulties that the DBA has had in administering the database. Also inquire about any proprietary languages, such as Oracle's PL*SQL, that are being used. Learn about possible differences in versions used at different locations and explore any difficulties that occur if different versions are installed.
3. Arrange an interview with a database administrator in your area who has at least seven years of experience as a database administrator. Focus the interview on understanding how DBA responsibilities and the way they are completed have changed during the DBA's tenure. Does the DBA have to generate more or less SQL code to administer the databases now than in the past? Has the position become more or less stressful?

References

Arvin, T. 2005. "Comparison of Different SQL Implementations" this and other information accessed at **http://troelsarvin.blogspot.com**.

Codd, E. F. 1970. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM* 13,6 (June): 77–87.

Date, C. J., and H. Darwen. 1997. *A Guide to the SQL Standard*. Reading, MA: Addison-Wesley.

Eisenberg, A., J. Melton, K. Kulkarni, J. E. Michels, and F. Zemke. 2004. "SQL:2003 Has Been Published." *SIGMOD Record* 33,1 (March):119–126.

Gorman, M. M. 2001. "Is SQL a Real Standard Anymore?" *The Data Administration Newsletter* (July), available at **www.tdan.com/i016hy01.htm**.

Lai, E. 2007. "IDC: Oracle Extended Lead Over IBM in 2006 Database Market." *Computerworld* (April 26), available at **www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9017898&intsrc=news_list**.

van der Lans, R. F. 2006. *Introduction to SQL; Mastering the Relational Database Language*, 4th ed. Workingham, UK: Addison-Wesley.

Further Reading

Bagui, S., and R. Earp. 2006. *Learning SQL on SQL Server 2005.* Sebastopol, CA: O'Reilly Media, Inc.

Bordoloi, B., and D. Bock. 2004. *Oracle SQL*. Upper Saddle River, NJ: Pearson Prentice Hall.

Celko, J. 2006. *Joe Celko's SQL Puzzles & Answers*, 2nd ed. San Francisco: Morgan Kaufmann.

Guerrero, F. G., and C. E. Rojas. 2001. *Microsoft SQL Server 2000 Programming by Example*. Indianapolis: QUE Corporation.

Gulutzan, P., and T. Petzer. 1999. *SQL-99 Complete, Really*. Lawrence, KS: R&D Books.

Nielsen, P. 2003. *Microsoft SQL Server 2000 Bible*. New York: Wiley Publishing, Inc.

Web Resources

http://standards.ieee.org The home page of the IEEE Standards Association.

http://troelsarvin.blogspot.com/ Blog that provides a detailed comparison of different SQL implementations, including DB2, Microsoft SQL, MySQL, Oracle, and PostGreSQL.

www.1keydata.com/sql/sql.html Web site that provides tutorials on a subset of ANSI standard SQL commands.

www.ansi.org Information on ANSI and the latest national and international standards.

www.coderecipes.net Web site that explains and shows examples for a wide range of SQL commands.

www.fluffycat.com/SQL/ Web site that defines a sample database and shows examples of SQL queries against this database.

www.incits.org The home page of the International Committee for Information Technology Standards, which used to be the National Committee for Information Technology Standards, which used to be the Accredited Standard Committee X3.

www.iso.ch International Organization for Standardization Web site, from which copies of current standards may be purchased.

www.itl.nist.gov/div897/ctg/dm/sql_examples.htm Web site that shows examples of SQL commands for creating tables and views, updating table contents, and performing some SQL database administration commands.

www.java2s.com/Code/SQL/CatalogSQL.htm Web site that provides tutorials on SQL in a MySQL environment.

www.mysql.com The official home page for MySQL, which includes many free downloadable components for working with MySQL.

www.paragoncorporation.com/ArticleDetail.aspx?ArticleID=27 Web site that provides a brief explanation of the power of SQL and a variety of sample SQL queries.

www.sqlcourse.com and www.sqlcourse2.com Web sites that provide tutorials for a subset of ANSI SQL, along with a practice database.

www.teradatastudentnetwork.com Web site where your instructor may have created some course environments for you to use Teradata SQL Assistant, Web Edition, with one or more of the Pine Valley Furniture and Mountain View Community Hospital data sets for this text.

www.tizag.com/sqlTutorial/ A set of tutorials on SQL concepts and commands.

www.wiscorp.com/SQLStandards.html Whitemarsh Information Systems Corp., a good source of information about SQL standards, including SQL:2003 and later standards.

CASE

Mountain View Community Hospital

Case Description

This case segment uses the physical designs you constructed for Mountain View Community Hospital (MVCH) in Chapter 5 to complete the case questions and case exercises.

Case Questions

1. What version of SQL and what RDBMS will you use to do the case exercises?
2. Which CASE tools are available for completing the case exercises? Can the CASE tool you are using generate the database schema from the physical data model(s) you created?
3. Can you suggest an easy way to populate your tables if you want to create a large set of test data?
4. How do the actual values you are using help you to test the functionality of your database?

Case Exercises

1. In Case Exercise 1 in Chapter 5, you created the physical data model for Dr. Z's database that keeps track of patients checking in. You may recall that Dr. Z decided to use SQL Server. Instructions for installing SQL Server and SQL Server Management Studio Express are available in the Pine Valley sample database area of this book's Web site.
   a. Using the design you created in Chapter 5, create the database and tables using SQL. Be sure to create the SQL assertions necessary to ensure referential integrity and other constraints.
   b. Populate the database with sample data. (MVCH Figure 4-5 in Chapter 4 provides some sample data, but you need a few more patients and visits for the queries in part c.)
   c. Write and test some queries that will work using your sample data. Write queries that
      i. Select information from only one of the tables (e.g., an alphabetical listing of all patients, an alphabetical listing of all the patients assigned to one of the social workers, etc.).
      ii. Aggregate information from one attribute in a table (e.g., How often has patient 8766 visited the MS Center at MVCH in a given month? How many patients are assigned to each social worker?).
      iii. Try out the various functions, such as MIN, MAX, and AVG (e.g., What is the average level of pain reported by Dr. Z's patients? What is the worst level of pain his patients have experienced?).
2. Using your database from Case Exercise 1, write and test SQL queries that
   a. Select information from only one of the tables.
   b. Aggregate information from one attribute in a table.
   c. Try out the various functions, such as MIN, MAX, and AVG.
   d. Qualify results by category.

Project Assignments

P1. Use the physical data model you created in Chapter 5 to guide you in writing the SQL statements for creating the MVCH database for the relational schema you created in Chapter 4.
   a. Write the SQL statements for creating the tables, specifying data types and field lengths, establishing primary keys and foreign keys, and implementing other constraints you identified.
   b. Following the examples in Chapter 5, write the SQL statements that create the indexes.

P2. Select a portion of your database and populate it with sample data. Be prepared to defend the sample test data that you insert into your database.

P3. Write and execute a variety of queries, based on the introduction to SQL in this chapter to test the functionality of your database. Ensure that your queries are correct and produce the results you expected.