Chapter 7

Introduction to Structured Query Language (SQL)

After completing this chapter, you will be able to:

- Retrieve specified columns of data from a database
- Join multiple tables in a single SQL query
- Restrict data retrievals to rows that match complex criteria
- Aggregate data across groups of rows
- Create subqueries to preprocess data for inclusion in other queries
- Identify and use a variety of SQL functions for string, numeric, and date manipulation
- Explain the key principles in crafting a SELECT query

Preview

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

Although it is quite useful and powerful, SQL is not meant to stand alone in the applications arena. Data entry with SQL is possible but awkward, as are data corrections and additions. SQL itself does not create menus, special report forms, overlays, pop-ups, or other features that end users usually expect. Instead, those features are available as vendor-supplied enhancements. SQL focuses on data definition (creating tables and indexes) and data manipulation (adding, modifying, deleting, and retrieving data). The most common task for SQL programmers is data retrieval. The ability to retrieve data from a database to satisfy business requirements is one of the most critical skills for database professionals. This chapter covers data retrieval in considerable detail.

Data Files and Available Formats

	MS Access	Oracle	MS SQL	My SQL		MS Access	Oracle	MS SQL	My SQL
CH07_SaleCo	✓	✓	✓	✓	CH07_ConstructCo	✓	✓	✓	✓
					CH07_LargeCo	✓	✓	✓	✓
					Ch07_Fact	✓	✓	✓	✓

Data Files Available on cengagebrain.com

7-1 Introduction to SQL

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

SQL functions fit into several broad categories:

- It is a *data manipulation language (DML)*. SQL includes commands to insert, update, delete, and retrieve data within the database tables. The data manipulation commands you will learn in this chapter are listed in Table 7.1. In this chapter, we will concentrate on the commands to retrieve data in interesting ways.
- It is a *data definition language (DDL)*. SQL includes commands to create database objects such as tables, indexes, and views, as well as commands to define access rights to those database objects. Some common data definition commands you will learn about in Chapter 8, Advanced SQL, are listed in Table 7.2.
- It is a *transaction control language (TCL)*. The DML commands in SQL are executed within the context of a **transaction**, which is a logical unit of work composed of one or more SQL statements, as defined by business rules (see Chapter 10, Transaction Management and Concurrency Control). SQL provides commands to control the processing of these statements an indivisible unit of work. These will be discussed in Chapter 8, after you learn about the DML commands that compose a transaction.
- It is a *data control language (DCL)*. Data control commands are used to control access to data objects, such as giving a one user permission to only view the PRODUCT table, and giving another use permission to change the data in the PRODUCT table. Common TCL and DCL commands are shown in Table 7.3.

SQL is relatively easy to learn. Its basic command set has a vocabulary of fewer than 100 words. Better yet, SQL is a nonprocedural language: you merely command *what* is to be done; you do not have to worry about *how*. For example, a single command creates the complex table structures required to store and manipulate data successfully; end users and programmers do not need to know the physical data storage format or the complex activities that take place when a SQL command is executed.

The American National Standards Institute (ANSI) prescribes a standard SQL. The ANSI SQL standards are also accepted by the International Organization for Standardization (ISO), a consortium composed of national standards bodies of more than 150 countries. Although adherence to the ANSI/ISO SQL standard is usually required in commercial and government contract database specifications, many RDBMS vendors add their own special enhancements. Consequently, it is seldom possible to move a SQL-based application from one RDBMS to another without making some changes.

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

7-1a Data Types

transaction
A logical unit of work composed of one or more SQL statements.

The ANSI/ISO SQL standard defines many different data types. A data type is a specification about the kinds of data that can be stored in an attribute. A more

TABLE 7.1

SQL DATA MANIPULATION COMMANDS

COMMAND, OPTION, OR OPERATOR	DESCRIPTION	COVERED
SELECT	**Selects attributes from rows in one or more tables or views**	**Chapter 7**
FROM	Specifies the tables from which data should be retrieved	Chapter 7
WHERE	Restricts the selection of rows based on a conditional expression	Chapter 7
GROUP BY	Groups the selected rows based on one or more attributes	Chapter 7
HAVING	Restricts the selection of grouped rows based on a condition	Chapter 7
ORDER BY	Orders the selected rows based on one or more attributes	Chapter 7
INSERT	**Inserts row(s) into a table**	**Chapter 8**
UPDATE	**Modifies an attribute's values in one or more table's rows**	**Chapter 8**
DELETE	**Deletes one or more rows from a table**	**Chapter 8**
Comparison operators		**Chapter 7**
=, <, >, <=, >=, <>, !=	Used in conditional expressions	Chapter 7
Logical operators		**Chapter 7**
AND/OR/NOT	Used in conditional expressions	Chapter 7
Special operators	**Used in conditional expressions**	**Chapter 7**
BETWEEN	Checks whether an attribute value is within a range	Chapter 7
IN	Checks whether an attribute value matches any value within a value list	Chapter 7
LIKE	Checks whether an attribute value matches a given string pattern	Chapter 7
IS NULL	Checks whether an attribute value is null	Chapter 7
EXISTS	Checks whether a subquery returns any rows	Chapter 7
DISTINCT	Limits values to unique values	Chapter 7
Aggregate functions	**Used with SELECT to return mathematical summaries on columns**	**Chapter 7**
COUNT	Returns the number of rows with non-null values for a given column	Chapter 7
MIN	Returns the minimum attribute value found in a given column	Chapter 7
MAX	Returns the maximum attribute value found in a given column	Chapter 7
SUM	Returns the sum of all values for a given column	Chapter 7
AVG	Returns the average of all values for a given column	Chapter 7

thorough discussion of data types will wait until Chapter 8, when we discuss the SQL commands to implement entities and attributes as tables and columns. However, a basic understanding of data types is needed before we can discuss how to retrieve data. Data types influence queries that retrieve data because there are slight differences in the syntax of SQL and how it behaves during a query that are based on the data type of the column being retrieved. For now, consider that there are three fundamental types of data: character data, numeric data, and date data. *Character data* is composed of any printable characters such as alphabetic values, digits, punctuation, and special characters. Character data is also often referred to as a "string" because it is a collection of characters threaded together to create the value. *Numeric data* is composed of digits, such that the data has a specific numeric value. *Date data* is composed of date and,

TABLE 7.2

SQL DATA DEFINITION COMMANDS

COMMAND OR OPTION	DESCRIPTION	COVERED
CREATE SCHEMA AUTHORIZATION	**Creates a database schema**	**Chapter 8**
CREATE TABLE	**Creates a new table in the user's database schema**	**Chapter 8**
NOT NULL	Ensures that a column will not have null values	Chapter 8
UNIQUE	Ensures that a column will not have duplicate values	Chapter 8
PRIMARY KEY	Defines a primary key for a table	Chapter 8
FOREIGN KEY	Defines a foreign key for a table	Chapter 8
DEFAULT	Defines a default value for a column (when no value is given)	Chapter 8
CHECK	Validates data in an attribute	Chapter 8
CREATE INDEX	**Creates an index for a table**	**Chapter 8**
CREATE VIEW	**Creates a dynamic subset of rows and columns from one or more tables**	**Chapter 8**
ALTER TABLE	**Modifies a table's definition (adds, modifies, or deletes attributes or constraints)**	**Chapter 8**
CREATE TABLE AS	**Creates a new table based on a query in the user's database schema**	**Chapter 8**
DROP TABLE	**Permanently deletes a table (and its data)**	**Chapter 8**
DROP INDEX	**Permanently deletes an index**	**Chapter 8**
DROP VIEW	**Permanently deletes a view**	**Chapter 8**

TABLE 7.3

OTHER SQL COMMANDS

COMMAND OR OPTION	DESCRIPTION	COVERED
Transaction Control Language		
COMMIT	Permanently saves data changes	Chapter 8
ROLLBACK	Restores data to its original values	Chapter 8
Data Control Language		
GRANT	Gives a user permission to take a system action or access a data object	Chapter 16
REVOKE	Removes a previously granted permission from a user	Chapter 16

occasionally, time values. Although character data may contain digits, the DBMS does not recognize the numeric value of those digits.

7-1b SQL Queries

At the heart of SQL is the query. In Chapter 1, Database Systems, you learned that a query is a spur-of-the-moment question. Actually, in the SQL environment, the word *query* covers both questions and actions. Most SQL queries are used to answer questions such as these: "What products currently held in inventory are priced over $100, and what is the quantity on hand for each of those products?" or "How many employees have been hired since January 1, 2016, by each of the company's departments?" However,

many SQL queries are used to perform actions such as adding or deleting table rows or changing attribute values within tables. Still other SQL queries create new tables or indexes. For a DBMS, a *query* is simply a SQL statement that must be executed. In most database-related jobs, retrieving data is by far the most common type of task. Not only do database professionals have to know how to retrieve data from the database, but virtually all application programmers need this skill as well.

Data retrieval is done in SQL using a SELECT query. When you run a SELECT command on a table, the RDBMS returns a set of one or more rows that have the same characteristics as a relational table. This is a very important characteristic of SQL commands. By default, most SQL data manipulation commands operate over an entire table (relation), which is why SQL commands are said to be *set-oriented* commands. A SQL **set-oriented** command works over a set of rows. The set may include one or more columns and zero or more rows from one or more tables. A SELECT query specifies which data should be retrieved and how it should be filtered, aggregated, and displayed. There are many potential clauses, or parts, to a SELECT query, as shown in Table 7.1. Constructing a SELECT query is similar to constructing objects with building blocks. The database programmer has to understand what each building block (clause) does and how the blocks fit together. Then he or she can make a plan for which blocks to use and determine how to assemble those blocks to produce the desired result.

set-oriented
Dealing with or related to sets, or groups of things. In the relational model, SQL operators are set-oriented because they operate over entire sets of rows and columns at once.

7-1c The Database Model

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

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

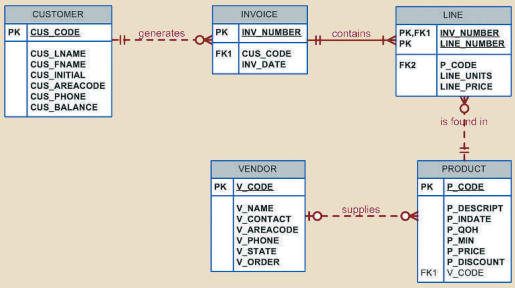
- A customer may generate many invoices. Each invoice is generated by one customer.
- An invoice contains one or more invoice lines. Each invoice line is associated with one invoice.
- Each invoice line references one product. A product may be found in many invoice lines. (You can sell more than one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

Except as noted, the database model shown in Figure 7.1 will be used for the queries in the remainder of the chapter. Recall that when an ERD is implemented as a database, each entity becomes a table in the database, and each attribute within an entity becomes a column in that table.

Online Content
The database model in Figure 7.1 is implemented in the Microsoft Access Ch07_SaleCo database, which is available at *www.cengagebrain.com*. (This database contains a few additional tables that are not reflected in Figure 7.1. These tables are used for discussion purposes only.) Scripts to create these tables in Oracle, MySQL, and SQL Server are also available at *www.cengagebrain.com*.

Note
This chapter focuses on SELECT queries to retrieve data from tables. Chapter 8 will explain how those tables are actually created and how the data is loaded into them. This reflects the experience of most entry-level database positions. As a new hire working with databases, you will likely spend quite a bit of time retrieving data from tables that already exist before you begin creating new tables and modifying the data.

FIGURE 7.1 THE DATABASE MODEL



7-2 Basic SELECT Queries

Each clause in a SELECT query performs a specific function. Understanding the function of each clause is key to developing the skills to construct queries to satisfy the reporting needs of the users. The following clauses will be covered in this chapter (although not in this order).

- SELECT—specifies the attributes to be returned by the query
- FROM—specifies the table(s) from which the data will be retrieved
- WHERE—filters the rows of data based on provided criteria
- GROUP BY—groups the rows of data into collections based on sharing the same values in one or more attributes
- HAVING—filters the groups formed in the GROUP BY clause based on provided criteria
- ORDER BY—sorts the final query result rows in ascending or descending order based on the values of one or more attributes.

Although SQL commands can be grouped together on a single line, complex command sequences are best shown on separate lines, with space between the SQL command and the command's components. Using that formatting convention makes it much easier to see the components of the SQL statements, which in turn makes it easy to trace the SQL logic and make corrections if necessary. The number of spaces used in the indention is up to you. For a SELECT query to retrieve data from the database, it will require at least a SELECT column list and a FROM clause. The **SELECT** column list specifies the relational projection, as discussed in Chapter 3, The Relational Database Model. The column list allows the programmer to specify which columns should be retrieved

SELECT
A SQL command that yields the values of all rows or a subset of rows in a table. The SELECT statement is used to retrieve data from tables.

by the query and the order in which they should be returned. Only columns specified in the column list will appear in the query result. The **FROM** clause is used to specify the table from which the data will be retrieved. It is common for queries to retrieve data from multiple tables that have been joined together, as discussed in Chapter 3. However, first, we will focus on things that can be done with the column list before we move on to the FROM clause options.

7-3 SELECT Statement Options

The SELECT query specifies the columns to be retrieved as a column list. The syntax for a basic SELECT query that retrieves data from a table is:

```
SELECT    columnlist
FROM      tablelist;
```

The *columnlist* represents one or more attributes, separated by commas. If the programmer wants all of the columns to be returned, then the asterisk (*) wildcard can be used. A **wildcard character** is a symbol that can be used as a general substitute for other characters or commands. This wildcard means "all columns." For example, the following query would return all of the data from the PRODUCT table (see Figure 7.2).

```
SELECT    *
FROM      PRODUCT;
```

FROM
A SQL clause that specifies the table or tables from which data is to be retrieved.

wildcard character
A symbol that can be used as a general substitute for: (1) all columns in a table (*) when used in an attribute list of a SELECT statement or (2) zero or more characters in a SQL LIKE clause condition (% and _).

FIGURE 7.2 SELECT AN ENTIRE TABLE

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

In this query, the column list indicates that all columns (and by default all of the rows) should be returned. The FROM clause specifies that the data from the PRODUCT table is to be used. Recall from Chapter 3 that projection does not limit the rows being returned. To limit the rows being returned, relational selection (or restriction) must be used. The column list allows the programmer to specify which columns should be returned, as shown in the next query (see Figure 7.3).


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

FIGURE 7.3 SELECT WITH A COLUMN LIST

P_CODE	P_DESCRIPT	P_PRICE	P_QOH
11QER/31	Power painter, 15 psi., 3-nozzle	109.99	8
13-Q2/P2	7.25-in. pwr. saw blade	14.99	32
14-Q1/L3	9.00-in. pwr. saw blade	17.49	18
1546-QQ2	Hrd. cloth, 1/4-in., 2x50	39.95	15
1558-QW1	Hrd. cloth, 1/2-in., 3x50	43.99	23
2232/QTY	B&D jigsaw, 12-in. blade	109.92	8
2232/QWE	B&D jigsaw, 8-in. blade	99.87	6
2238/QPD	B&D cordless drill, 1/2-in.	38.95	12
23109-HB	Claw hammer	9.95	23
23114-AA	Sledge hammer, 12 lb.	14.40	8
54778-2T	Rat-tail file, 1/8-in. fine	4.99	43
89-WRE-Q	Hicut chain saw, 16 in.	256.99	11
PVC23DRT	PVC pipe, 3.5-in., 8-ft	5.87	188
SM-18277	1.25-in. metal screw, 25	6.99	172
SW-23116	2.5-in. wd. screw, 50	8.45	237
WR3/TT3	Steel matting, 4'x8'x1/6", .5" mesh	119.95	18

This query specifies that the data should come from the PRODUCT table, and that only the product code, description, price, and quantity on hand columns should be included. Notice that only the requested columns are returned and that the columns are in the same order in the output as they were listed in the query. To display the columns in a different order, simply change the order of the columns in the column list.

7-3a Using Column Aliases

Recall that the attribute within an entity is implemented as a column in the table. The attribute name becomes the name of that column. When that column is retrieved in a query, the attribute name is used as a label, or column heading, in the query output by default. If the programmer wants a different name to be used as the label in the output, a new name can be specified. The new name is referred to as an **alias**. For example, aliases are used in the following query (see Figure 7.4).

```
SELECT    P_CODE, P_DESCRIPT AS DESCRIPTION, P_PRICE AS "Unit Price",
          P_QOH QTY
FROM      PRODUCT;
```

In this query and its output in Figure 7.4, the DESCRIPT attribute P_ is given the alias DESCRIPTION, P_PRICE is given the alias Unit Price, and P_QOH is given the alias QTY. There are a few things of interest about the use of these aliases:

- Not all columns in a query must use an alias
- AS is optional, but recommended
- Aliases that contain a space must be inside a delimiter (quotes)

alias
An alternative name for a column or table in a SQL statement.

FIGURE 7.4 SELECT WITH COLUMN ALIASES

P_CODE	DESCRIPTION	Unit Price	QTY
11QER/31	Power painter, 15 psi., 3-nozzle	109.99	8
13-Q2/P2	7.25-in. pwr. saw blade	14.99	32
14-Q1/L3	9.00-in. pwr. saw blade	17.49	18
1546-QQ2	Hrd. cloth, 1/4-in., 2x50	39.95	15
1558-QW1	Hrd. cloth, 1/2-in., 3x50	43.99	23
2232/QTY	B&D jigsaw, 12-in. blade	109.92	8
2232/QWE	B&D jigsaw, 8-in. blade	99.87	6
2238/QPD	B&D cordless drill, 1/2-in.	38.95	12
23109-HB	Claw hammer	9.95	23
23114-AA	Sledge hammer, 12 lb.	14.40	8
54778-2T	Rat-tail file, 1/8-in. fine	4.99	43
89-WRE-Q	Hicut chain saw, 16 in.	256.99	11
PVC23DRT	PVC pipe, 3.5-in., 8-ft	5.87	188
SM-18277	1.25-in. metal screw, 25	6.99	172
SW-23116	2.5-in. wd. screw, 50	8.45	237
WR3/TT3	Steel matting, 4'x8'x1/6", .5" mesh	119.95	18

The AS keyword is not required, but it is recommended. If there is a space between the column name and the alias, the DBMS will interpret the alias correctly. However, as we shall soon see, it is possible to embed formulas and functions within the column list, and you will generally want an alias for the columns produced. In those cases, having the AS keyword makes it much easier to read the query and understand that the alias is just an alias and not a part of the formula. Finally, the DBMS expects an alias to appear as a single word. If there are any spaces in the alias, then the programmer must use a delimiter to indicate where the alias begins and ends. In Figure 7.4, a double-quote delimiter was used around the Unit Price alias because it contains a space. Most DBMS products allow double quotes around a column alias.

Note

Using delimiters with column aliases even when the alias does not contain a space can serve other purposes. In some DBMSs, if the column alias is not placed inside a delimiter, it is automatically converted to uppercase letters. In those cases, using the delimiter allows the programmer to control the capitalization of the column alias. Using delimiters also allows a column alias to contain a special character, such as "+", or a SQL keyword, such as "SELECT." In general, using special characters and SQL keywords in column aliases is discouraged, but it is possible.

Note

MySQL uses a special delimiter, the back tick " ` " (usually found to the left of the number 1 on a standard keyboard) as a delimiter for column aliases if you want to refer to that alias elsewhere within the query, such as the ORDER BY clause covered later in this chapter.

7-3b Using Computed Columns

A computed column (also called a calculated column) represents a derived attribute, as discussed in Chapter 4, Entity Relationship Modeling. Recall from Chapter 4 that a derived attribute may or may not be stored in the database. If the decision is made not to store the derived attribute, then the attribute must be calculated when it is needed. For example, suppose that you want to determine the total value of each of the products currently held in inventory. Logically, that determination requires the multiplication of each product's quantity on hand by its current price. You can accomplish this task with the following command:

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

Entering the SQL command generates the output shown in Figure 7.5.

FIGURE 7.5 SELECT STATEMENT WITH A COMPUTED COLUMN

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

SQL accepts any valid expressions (or formulas) in the computed columns. Such formulas can contain any valid mathematical operators and functions that are applied to attributes in any of the tables specified in the FROM clause of the SELECT statement. Different DBMS products vary in the column headings that are displayed for the computed column.

Note

MS Access automatically adds an Expr label to all computed columns when an alias is not specified. (The first computed column would be labeled Expr1; the second, Expr2; and so on.) Oracle uses the actual formula text as the label for the computed column. Other DBMSs return the column without a heading label.

To make the output more readable, an alias is typically used for any computed fields. For example, you can rewrite the previous SQL statement as follows:

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

The output of the command is shown in Figure 7.6.

FIGURE 7.6 SELECT STATEMENT WITH A COMPUTED COLUMN AND AN ALIAS

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

7-3c Arithmetic Operators: The Rule of Precedence

As you saw in the previous example, you can use arithmetic operators with table attributes in a column list or in a conditional expression. In fact, SQL commands are often used in conjunction with the arithmetic operators shown in Table 7.4.

TABLE 7.4

THE ARITHMETIC OPERATORS

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

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

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

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

rules of precedence
Basic algebraic rules that specify the order in which operations are performed. For example, operations within parentheses are executed first, so in the equation 2 + (3 × 5), the multiplication portion is calculated first, making the correct answer 17.

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

7-3d Date Arithmetic

Date data in the column list can be interesting when used in computed fields. Internally, the DBMS stores a date value in a numeric format. Although the details can be complicated, essentially, a date is stored as a day number, that is, the number of days that have passed since some defined point in history. Exactly what that point in history is varies from one DBMS to another. However, because the values are stored as a number of days, it is possible to perform date arithmetic in a query. For example, if today's date in some DBMS is the day number "250,000," then tomorrow will be "250,001," and yesterday was "249,999." Adding or subtracting a number from a date that is stored in a date data type returns the date that is the specified number of days from the given date. Subtracting one date value from another yields the number of days between those dates.

Suppose that a manager wants a list of all products, the dates they were received, and the warranty expiration date (90 days from receiving the product). To generate that list, you would make the following query:

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

This query uses a computed column with an alias and date arithmetic in a single query. The DBMS also has a function to return the current date on the database server, making it possible to write queries that reference the current date without having to change the contents of the query each day. For example, the DATE(), GETDATE(), and CURDATE() functions in MS Access, SQL Server, and MySQL, respectively, and the SYSDATE keyword in Oracle will all retrieve the current date. If a manager wants to a list of products and the warranty cutoff date for products, the query in Oracle would be:

```
SELECT    P_CODE, P_INDATE, SYSDATE – 90 AS CUTOFF
FROM      PRODUCT;
```

In this query, the output would change based on the current date. You can use these functions anywhere a date literal is expected.

7-3e Listing Unique Values

How many *different* vendors are currently represented in the PRODUCT table? A simple listing (SELECT) is not very useful if the table contains several thousand rows and you have to sift through the vendor codes manually. Fortunately, SQL's **DISTINCT** clause produces a list of only those values that are different from one another. For example, the command

```
SELECT    DISTINCT V_CODE
FROM      PRODUCT;
```

yields only the different vendor codes (V_CODE) in the PRODUCT table, as shown in Figure 7.7. The DISTINCT keyword only appears once in the query, and that is immediately following the SELECT keyword. Note that the first output row shows a null. Rows may contain a null for the V_CODE attribute if the product is developed in-house or if it is purchased directly from the manufacturer. As discussed in Chapter 3, nulls can be problematic because it is difficult to know what the null means in the business environment. Nulls can also be problematic when writing SQL code. Different operators and functions treat nulls differently. For example, the DISTINCT keyword considers

DISTINCT
A SQL clause that produces only a list of values that are different from one another.

null to be a value, and it considers all nulls to be the same value. In later sections, we will encounter functions that ignore nulls, and we will see comparisons that consider all nulls to be different. As a SQL developer, you must understand how nulls will be treated by the code you are writing.

FIGURE 7.7 A LISTING OF DISTINCT V_CODE VALUES IN THE PRODUCT TABLE

V_CODE
21225
21231
21344
23119
24288
25595

7-4 FROM Clause Options

The **FROM** clause of the query specifies the table or tables from which the data is to be retrieved. In the following query, the data is being retrieved from only the PRODUCT table.

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

In practice, most SELECT queries will need to retrieve data from multiple tables. In Chapter 3, we looked at JOIN operators that are used to combine data from multiple tables in meaningful ways. The database design process that led to the current database was in many ways a process of decomposition—the designer took an integrated set of data related to a business problem and decomposed that data into separate entities to create a flexible, stable structure for storing and manipulating that data. Now, through the use of joins, the programmer reintegrates pieces of the data to satisfy the users' information needs. *Inner joins* return only rows from the tables that match on a common value. *Outer joins* return the same matched rows as the inner join, plus unmatched rows from one table or the other. (The various types of joins are presented in Chapter 3.)

The join condition is generally composed of an equality comparison between the foreign key and the primary key of related tables. For example, suppose that you want to join the two tables VENDOR and PRODUCT. Because V_CODE is the foreign key in the PRODUCT table and the primary key in the VENDOR table, the link is established on V_CODE. (See Table 7.5.)

TABLE 7.5
CREATING LINKS THROUGH FOREIGN KEYS

TABLE	ATTRIBUTES TO BE SHOWN	LINKING ATTRIBUTE
PRODUCT	P_DESCRIPT, P_PRICE	V_CODE
VENDOR	V_NAME, V_CONTACT, V_AREACODE, V_PHONE	V_CODE

Joining the PRODUCT and VENDOR tables, which produces the output shown in Figure 7.8, can be accomplished in multiple ways.

FIGURE 7.8 THE RESULTS OF A JOIN

P_DESCRIPT	P_PRICE	V_NAME	V_CONTACT	V_AREACODE	V_PHONE
Claw hammer	9.95	Bryson, Inc.	Smithson	615	223-3234
1.25-in. metal screw, 25	6.99	Bryson, Inc.	Smithson	615	223-3234
2.5-in. wd. screw, 50	8.45	D&E Supply	Singh	615	228-3245
7.25-in. pwr. saw blade	14.99	Gomez Bros.	Ortega	615	889-2546
9.00-in. pwr. saw blade	17.49	Gomez Bros.	Ortega	615	889-2546
Rat-tail file, 1/8-in. fine	4.99	Gomez Bros.	Ortega	615	889-2546
Hrd. cloth, 1/4-in., 2x50	39.95	Randsets Ltd.	Anderson	901	678-3998
Hrd. cloth, 1/2-in., 3x50	43.99	Randsets Ltd.	Anderson	901	678-3998
B&D jigsaw, 12-in. blade	109.92	ORDVA, Inc.	Halsford	615	898-1234
B&D jigsaw, 8-in. blade	99.87	ORDVA, Inc.	Halsford	615	898-1234
Hicut chain saw, 16 in.	256.99	ORDVA, Inc.	Halsford	615	898-1234
Power painter, 15 psi., 3-nozzle	109.99	Rubicon Systems	Orton	904	456-0092
B&D cordless drill, 1/2-in.	38.95	Rubicon Systems	Orton	904	456-0092
Steel matting, 4'x8'x1/6", .5" mesh	119.95	Rubicon Systems	Orton	904	456-0092

7-4a Natural Join

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

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

The natural join performs the following tasks:

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

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

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

The results of this query are shown in Figure 7.9.

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

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


FIGURE 7.9 CUSTOMER NATURAL JOIN INVOICE RESULTS

CUS_CODE	CUS_LNAME	INV_NUMBER	INV_DATE
10011	Dunne	1002	16-Jan-18
10011	Dunne	1004	17-Jan-18
10011	Dunne	1008	17-Jan-18
10012	Smith	1003	16-Jan-18
10014	Orlando	1001	16-Jan-18
10014	Orlando	1006	17-Jan-18
10015	O'Brian	1007	17-Jan-18
10018	Farriss	1005	17-Jan-18

The results of this SQL code are shown in Figure 7.10.

FIGURE 7.10 NATURAL JOIN WITH THREE TABLES

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

Note

While some DBMS include the NATURAL JOIN operator, it is generally discouraged in practice because it can be unclear to the programmer and to others performing maintenance on the code exactly which attribute or attributes the DBMS is using as the common attribute to perform the join. Even if the DBMS is correctly joining the tables when the code is originally written, subsequent changes to the structure of the database tables being used can cause the DBMS to join the tables incorrectly at a later point in time.

7-4b JOIN USING Syntax

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

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

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

```
SELECT    P_CODE, P_DESCRIPT, V_CODE, V_NAME, V_AREACODE, V_PHONE
FROM      PRODUCT JOIN VENDOR USING (V_CODE);
```

The SQL statement produces the results shown in Figure 7.11.

FIGURE 7.11 JOIN USING RESULTS

P_CODE	P_DESCRIPT	V_CODE	V_NAME	V_AREACODE	V_PHONE
23109-HB	Claw hammer	21225	Bryson, Inc.	615	223-3234
SM-18277	1.25-in. metal screw, 25	21225	Bryson, Inc.	615	223-3234
SW-23116	2.5-in. wd. screw, 50	21231	D&E Supply	615	228-3245
13-Q2/P2	7.25-in. pwr. saw blade	21344	Gomez Bros.	615	889-2546
14-Q1/L3	9.00-in. pwr. saw blade	21344	Gomez Bros.	615	889-2546
54778-2T	Rat-tail file, 1/8-in. fine	21344	Gomez Bros.	615	889-2546
1546-QQ2	Hrd. cloth, 1/4-in., 2x50	23119	Randsets Ltd.	901	678-3998
1558-QW1	Hrd. cloth, 1/2-in., 3x50	23119	Randsets Ltd.	901	678-3998
2232/QTY	B&D jigsaw, 12-in. blade	24288	ORDVA, Inc.	615	898-1234
2232/QWE	B&D jigsaw, 8-in. blade	24288	ORDVA, Inc.	615	898-1234
89-WRE-Q	Hicut chain saw, 16 in.	24288	ORDVA, Inc.	615	898-1234
11QER/31	Power painter, 15 psi., 3-nozzle	25595	Rubicon Systems	904	456-0092
2238/QPD	B&D cordless drill, 1/2-in.	25595	Rubicon Systems	904	456-0092
WR3/TT3	Steel matting, 4'x8'x1/6", .5" mesh	25595	Rubicon Systems	904	456-0092

The preceding SQL command sequence joins a row in the PRODUCT table with a row in the VENDOR table, in which the V_CODE values of these rows are the same, as indicated in the USING clause. Because any vendor can deliver any number of ordered products, the PRODUCT table might contain multiple V_CODE entries for each V_CODE entry in the VENDOR table. In other words, each V_CODE in VENDOR can be matched with many V_CODE rows in PRODUCT.

As with the NATURAL JOIN command, the JOIN USING operand does not require table qualifiers and only returns one copy of the common attribute.

Note

Oracle and MySQL support the JOIN USING syntax. MS SQL Server and Access do not. If JOIN USING is used in Oracle, then table qualifiers cannot be used with the common attribute anywhere within the query. MySQL allows table qualifiers on the common attribute anywhere except in the USING clause itself.

7-4c JOIN ON Syntax

The previous two join styles use common attribute names in the joining tables. Another way to express a join when the tables have no common attribute names is to use the JOIN ON operand. The query returns only the rows that meet the indicated join condition. The join condition typically includes an equality comparison expression of two columns. (The columns may or may not share the same name, but obviously they must have comparable data types.) The syntax is:

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

The following example performs a join of the INVOICE and LINE tables using the ON clause. The result is shown in Figure 7.12.

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

FIGURE 7.12 JOIN ON RESULTS

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

Note

Best practices for SQL programming suggest that JOIN ON or JOIN USING should be used instead of NATURAL JOIN or old-style joins, discussed later in this chapter. JOIN USING syntax is not as widely supported among DBMS vendors and it requires that the common attributes have exactly the same name in the tables being joined. As a result, the opportunities to use it are more limited than JOIN ON, which is widely supported and has no limitations on the common attributes. Therefore, in practice, JOIN ON is typically considered the join syntax of preference.

7-4d Common Attribute Names

One of the characteristics of a relational table presented in Chapter 3 is that no two columns in a table can have exactly the same name. Joining tables merges the rows in the tables using the specified join criteria to create a new, single table. In the process of combining these tables, not only are the rows merged but the columns of the tables are also placed together in the new table. As a result, even if each of the original tables had unique column names, it is likely that there are duplicate column names across the tables. When these columns are all placed in the same table by the join operation, it is possible to end up with duplicate column names in the resulting table. To enforce the relational requirement of unique column names in a table, the RDBMS will prefix the table names onto the column names. These fully qualified names typically do not display the table name qualifier in query results, but the query code must make use of the fully qualified names. The most common cause of duplicate column names is the existence of a foreign key. In fact, most queries will join tables using PK/FK combinations as the common attribute for the join criteria. The NATURAL JOIN and JOIN USING operands automatically eliminate duplicate columns for the common attribute to avoid the issue of duplicate column names. The JOIN ON clause does not automatically remove a copy of the common attribute, so it requires a table qualifier whenever the query references the common attribute. Notice the difference in the following code:

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

Produces the same result as (see Figure 7.13):

```
SELECT  P_CODE, V_CODE, V_NAME
FROM    PRODUCT JOIN VENDOR USING (V_CODE);
```

7-4e Outer Joins

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

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

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

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

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


The preceding SQL code and its results are shown in Figure 7.13.

FIGURE 7.13 LEFT JOIN RESULTS

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

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

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

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

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

The SQL code and its output are shown in Figure 7.14.

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

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

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

FIGURE 7.14 RIGHT JOIN RESULTS

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

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

The SQL code and its results are shown in Figure 7.15.

FIGURE 7.15 FULL OUTER JOIN RESULTS

P_CODE	V_CODE	V_NAME
	21226	SuperLoo, Inc.
	22567	Dome Supply
	24004	Brackman Bros.
	25443	B&K, Inc.
	25501	Damal Supplies
11QER/31	25595	Rubicon Systems
13-Q2/P2	21344	Gomez Bros.
14-Q1/L3	21344	Gomez Bros.
1546-QQ2	23119	Randsets Ltd.
1558-QW1	23119	Randsets Ltd.
2232/QTY	24288	ORDVA, Inc.
2232/QWE	24288	ORDVA, Inc.
2238/QPD	25595	Rubicon Systems
23109-HB	21225	Bryson, Inc.
23114-AA		
54778-2T	21344	Gomez Bros.
89-WRE-Q	24288	ORDVA, Inc.
PVC23DRT		
SM-18277	21225	Bryson, Inc.
SW-23116	21231	D&E Supply
WR3/TT3	25595	Rubicon Systems

Note
Oracle and MS SQL Server support the FULL JOIN syntax. MySQL and Access do not.

7-4f Cross Join

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

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

For example, the following command:

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

performs a cross join of the INVOICE and LINE tables that generates 144 rows. (There are 8 invoice rows and 18 line rows, yielding 8 × 18 = 144 rows.)

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

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

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

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

Note
Unlike Oracle, MS SQL Server, and MySQL, Access does not support the CROSS JOIN operator. However, all DBMS support producing a cross join by placing a comma between the tables in the FROM clause, which is the more common method for producing a cross join.

Note
Despite the name, CROSS JOIN is not truly a join operation since it does not unite the rows of the tables based on a common attribute.

cross join
A join that performs a relational product (or Cartesian product) of two tables.

7-4g Joining Tables with an Alias

An alias may be used to identify the source table from which the data is taken. The aliases P and V are used to label the PRODUCT and VENDOR tables in the next command sequence. Any legal table name may be used as an alias. (Also notice that there are no table name prefixes because the attribute listing contains no duplicate names in the SELECT statement.)

```
SELECT    P_DESCRIPT, P_PRICE, V_NAME, V_CONTACT, V_AREACODE,
          V_PHONE
FROM      PRODUCT P JOIN VENDOR V ON P.V_CODE = V.V_CODE;
```


Note
MS Access requires the AS keyword before a table alias. Oracle and MySQL do not use the AS keyword for a table alias, while MS SQL Server will accept table aliases either with or without the AS keyword. Using the AS keyword would change the above query to:

```
SELECT    P_DESCRIPT, P_PRICE, V_NAME, V_CONTACT, V_AREACODE,
          V_PHONE
FROM      PRODUCT AS P JOIN VENDOR AS V ON P.V_CODE = V.V_CODE;
```

The ability to specify a table alias is very useful. As you've seen, an alias can be used to shorten a table name within a query, but this is not the most common reason to use a table alias. The data models presented in most classes tend to be rather small, with at most a dozen or so tables. In practice, data models are often much larger. The authors have worked with companies that have data models with over 30,000 tables each! As you can imagine, when there are that many tables dealing with a business subject area, it becomes increasingly difficult for even a creative team of database designers to devise meaningful, descriptive entity names. As a result cryptic, abbreviation-filled entity names dominate many parts of the model. Using a table alias allows the database programmer to improve the maintainability of the code by using a table alias that is descriptive of what data the table is providing within the query. For example, in a healthcare industry data model that has twenty different tables of patient-related data and multiple tables dealing with a variety of policy, insurance, and employee exemptions, a table name named PDEPINPCEX that contains patient-dependent insurance cover policy exemptions can be given an alias-like EXEMPTS in a query. This greatly improves the readability of the query by replacing a table name that is not readily understandable with an alias that is.

7-4h Recursive Joins

A table alias is especially useful when a table must be joined to itself in a **recursive query**, as is the case when working with unary relationships. For example, suppose that you are working with the EMP table shown in Figure 7.16.

recursive query
A query that joins a table to itself.

FIGURE 7.16 CONTENTS OF THE EMP TABLE

EMP_NUM	EMP_TITLE	EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_DOB	EMP_HIRE_DATE	EMP_AREACODE	EMP_PHONE	EMP_MGR
100	Mr.	Kolmycz	George	D	15-Jun-42	15-Mar-85	615	324-5456	
101	Ms.	Lewis	Rhonda	G	19-Mar-65	25-Apr-86	615	324-4472	100
102	Mr.	Vandam	Rhett		14-Nov-58	20-Dec-90	901	675-8993	100
103	Ms.	Jones	Anne	M	16-Oct-74	28-Aug-94	615	898-3456	100
104	Mr.	Lange	John	P	08-Nov-71	20-Oct-94	901	504-4430	105
105	Mr.	Williams	Robert	D	14-Mar-75	08-Nov-98	615	890-3220	
106	Mrs.	Smith	Jeanine	K	12-Feb-68	05-Jan-89	615	324-7883	105
107	Mr.	Diante	Jorge	D	21-Aug-74	02-Jul-94	615	890-4567	105
108	Mr.	Wiesenbach	Paul	R	14-Feb-66	18-Nov-92	615	897-4358	
109	Mr.	Smith	George	K	18-Jun-61	14-Apr-89	901	504-3339	108
110	Mrs.	Genkazi	Leighla	W	19-May-70	01-Dec-90	901	569-0093	108
111	Mr.	Washington	Rupert	E	03-Jan-66	21-Jun-93	615	890-4925	105
112	Mr.	Johnson	Edward	E	14-May-61	01-Dec-83	615	898-4387	100
113	Ms.	Smythe	Melanie	P	15-Sep-70	11-May-99	615	324-9006	105
114	Ms.	Brandon	Marie	G	02-Nov-56	15-Nov-79	901	882-0845	108
115	Mrs.	Saranda	Hermine	R	25-Jul-72	23-Apr-93	615	324-5505	105
116	Mr.	Smith	George	A	08-Nov-65	10-Dec-88	615	890-2984	108

Using the data in the EMP table, you can generate a list of all employees with their managers' names by joining the EMP table to itself. In that case, you would also use aliases to differentiate the table from itself. The SQL command sequence would look like this:

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

The output of the preceding command sequence is shown in Figure 7.17.

FIGURE 7.17 USING AN ALIAS TO JOIN A TABLE TO ITSELF

EMP_NUM	E.EMP_LNAME	EMP_MGR	M.EMP_LNAME
112	Johnson	100	Kolmycz
103	Jones	100	Kolmycz
102	Vandam	100	Kolmycz
101	Lewis	100	Kolmycz
115	Saranda	105	Williams
113	Smythe	105	Williams
111	Washington	105	Williams
107	Diante	105	Williams
106	Smith	105	Williams
104	Lange	105	Williams
116	Smith	108	Wiesenbach
114	Brandon	108	Wiesenbach
110	Genkazi	108	Wiesenbach
109	Smith	108	Wiesenbach

7-5 ORDER BY Clause Options

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

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

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

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

The output is shown in Figure 7.18. Note that ORDER BY yields an ascending price listing.

Comparing the listing in Figure 7.18 to the actual table contents shown earlier in Figure 7.2, you will see that the lowest-priced product is listed first in Figure 7.18, followed by the next lowest-priced product, and so on. However, although ORDER BY produces a sorted output, the actual table contents are unaffected by the ORDER BY operation.

You can add DESC after the attribute to indicate descending order. To produce the listing with products sorted in descending order by price, you would enter:

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

ORDER BY
A SQL clause that is useful for ordering the output of a SELECT query (e.g., in ascending or descending order).

FIGURE 7.18 PRODUCTS SORTED BY PRICE IN ASCENDING ORDER

P_CODE	P_DESCRIPT	P_QOH	P_PRICE
54778-2T	Rat-tail file, 1/8-in. fine	43	4.99
PVC23DRT	PVC pipe, 3.5-in., 8-ft	188	5.87
SM-18277	1.25-in. metal screw, 25	172	6.99
SW-23116	2.5-in. wd. screw, 50	237	8.45
23109-HB	Claw hammer	23	9.95
23114-AA	Sledge hammer, 12 lb.	8	14.40
13-Q2/P2	7.25-in. pwr. saw blade	32	14.99
14-Q1/L3	9.00-in. pwr. saw blade	18	17.49
2238/QPD	B&D cordless drill, 1/2-in.	12	38.95
1546-QQ2	Hrd. cloth, 1/4-in., 2x50	15	39.95
1558-QW1	Hrd. cloth, 1/2-in., 3x50	23	43.99
2232/QWE	B&D jigsaw, 8-in. blade	6	99.87
2232/QTY	B&D jigsaw, 12-in. blade	8	109.92
11QER/31	Power painter, 15 psi., 3-nozzle	8	109.99
WR3/TT3	Steel matting, 4'x8'x1.6", .5" mesh	18	119.95
89-WRE-Q	Hicut chain saw, 16 in.	11	256.99

Ordered listings are used frequently. For example, suppose that you want to create a phone directory. It would be helpful if you could produce an ordered sequence (last name, first name, initial) in three stages:

1. ORDER BY last name.
2. Within matching last names, ORDER BY first name.
3. Within matching first and last names, ORDER BY middle initial.

Such a multilevel ordered sequence is known as a **cascading order sequence**, and it can be created easily by listing several attributes, separated by commas, after the ORDER BY clause.

The cascading order sequence is the basis for any telephone directory. To illustrate a cascading order sequence, use the following SQL command on the EMPLOYEE table:

```
SELECT    EMP_LNAME, EMP_FNAME, EMP_INITIAL, EMP_AREACODE,
          EMP_PHONE
FROM      EMPLOYEE
ORDER BY  EMP_LNAME, EMP_FNAME, EMP_INITIAL;
```

This command yields the results shown in Figure 7.19.

FIGURE 7.19 TELEPHONE LIST QUERY RESULTS

EMP_LNAME	EMP_FNAME	EMP_INITIAL	EMP_AREACODE	EMP_PHONE
Brandon	Marie	G	901	882-0845
Diante	Jorge	D	615	890-4567
Genkazi	Leighla	W	901	569-0093
Johnson	Edward	E	615	898-4387
Jones	Anne	M	615	898-3456
Kolmycz	George	D	615	324-5456
Lange	John	P	901	504-4430
Lewis	Rhonda	G	615	324-4472
Saranda	Hermine	R	615	324-5505
Smith	George	A	615	890-2984
Smith	George	K	901	504-3339
Smith	Jeanine	K	615	324-7883
Smythe	Melanie	P	615	324-9006
Vandam	Rhett		901	675-8993
Washington	Rupert	E	615	890-4925
Wiesenbach	Paul	R	615	897-4358
Williams	Robert	D	615	890-3220

cascading order sequence
A nested ordering sequence for a set of rows, such as a list in which all last names are alphabetically ordered and, within the last names, all first names are ordered.

The ORDER BY clause is useful in many applications, especially because the DESC qualifier can be invoked. For example, listing the most recent items first is a standard procedure. Typically, invoice due dates are listed in descending order. Or, if you want to examine budgets, it is probably useful to list the largest budget line items first.

You can use the ORDER BY clause in conjunction with other SQL operations, too. For example, note the use of a derived attribute in the following command sequence:

```
SELECT      P_CODE, P_DESCRIPT, V_CODE, P_PRICE * P_QOH AS TOTAL
FROM        PRODUCT
ORDER BY    V_CODE, TOTAL DESC;
```

The output is shown in Figure 7.20. The query results are sorted in ascending order by V_CODE, and then within matching vendor code values, the results are sorted in descending order by the derived total value attribute.

FIGURE 7.20 ORDERING BY A DERIVED ATTRIBUTE

P_CODE	P_DESCRIPT	V_CODE	TOTAL
PVC23DRT	PVC pipe, 3.5-in., 8-ft		1103.56
23114-AA	Sledge hammer, 12 lb.		115.20
SM-18277	1.25-in. metal screw, 25	21225	1202.28
23109-HB	Claw hammer	21225	228.85
SW-23116	2.5-in. wd. screw, 50	21231	2002.65
13-Q2/P2	7.25-in. pwr. saw blade	21344	479.68
14-Q1/L3	9.00-in. pwr. saw blade	21344	314.82
54778-2T	Rat-tail file, 1/8-in. fine	21344	214.57
1558-QW1	Hrd. cloth, 1/2-in., 3x50	23119	1011.77
1546-QQ2	Hrd. cloth, 1/4-in., 2x50	23119	599.25
89-WRE-Q	Hicut chain saw, 16 in.	24288	2826.89
2232/QTY	B&D jigsaw, 12-in. blade	24288	879.36
2232/QWE	B&D jigsaw, 8-in. blade	24288	599.22
WR3/TT3	Steel matting, 4'x8'x1/6", .5" mesh	25595	2159.10
11QER/31	Power painter, 15 psi., 3-nozzle	25595	879.92
2238/QPD	B&D cordless drill, 1/2-in.	25595	467.40

Note

If the ordering column has nulls, they are either first or last, depending on the RDBMS. Oracle supports adding a NULLS FIRST or NULLS LAST option to change the sort behavior of nulls in the ORDER BY clause. For example, the following command in Oracle returns the vendor codes sorted from largest to smallest but with the null vendor codes appearing last in the list.

```
SELECT      V_CODE, P_DESCRIPT
FROM        PRODUCT
ORDER BY    V_CODE DESC NULLS LAST;
```


7-6 WHERE Clause Options

In this section, you learn how to fine-tune the SELECT command by adding restrictions to the search criteria. When coupled with appropriate search conditions, SELECT is an incredibly powerful tool that enables you to transform data into information. For example, you can create queries that can answer questions such as these: "What products were supplied by a particular vendor?," "Which products are priced below $10?," and "How many products supplied by a given vendor were sold between January 5, 2018, and March 20, 2018?"

7-6a Selecting Rows with Conditional Restrictions

You can select partial table contents by placing restrictions on the rows to be included in the output. Use the **WHERE** clause to add conditional restrictions to the SELECT statement that limit the rows returned by the query. The following syntax enables you to specify which rows to select:

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

The SELECT statement retrieves all rows that match the specified condition(s)—also known as the conditional criteria—you specified in the WHERE clause. The *conditionlist* in the WHERE clause of the SELECT statement is represented by one or more conditional expressions, separated by logical operators. The WHERE clause is optional. If no rows match the specified criteria in the WHERE clause, you see a blank screen or a message that tells you no rows were retrieved. For example, consider the following query:

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

This query returns the description, quantity on hand, price, and vendor code for products with a vendor code of 21344, as shown in Figure 7.21.

FIGURE 7.21 SELECTED PRODUCT ATTRIBUTES FOR VENDOR CODE 21344

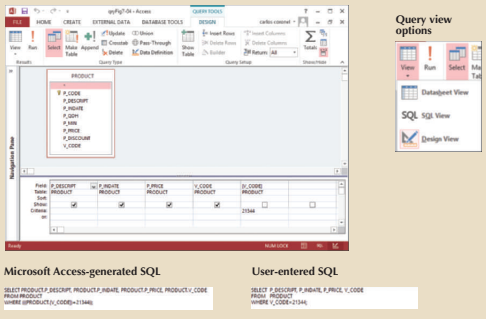
P_DESCRIPT	P_QOH	P_PRICE	V_CODE
7.25-in. pwr. saw blade	32	14.99	21344
9.00-in. pwr. saw blade	18	17.49	21344
Rat-tail file, 1/8-in. fine	43	4.99	21344

MS Access users can use the Access QBE (query by example) query generator to create the code throughout this chapter. However, as the code becomes more complex, these users may notice that code generated by Access becomes more and more significantly different than what is being presented in the chapter. This is because the Access QBE generates its own "native" version of SQL. You can also elect to type standard SQL

WHERE
A SQL clause that adds conditional restrictions to a SELECT statement that limit the rows returned by the query.

in the Access SQL window, as shown at the bottom of Figure 7.22. The figure shows the Access QBE screen, the SQL window's QBE-generated SQL, and the listing of the modified SQL.

FIGURE 7.22 THE MICROSOFT ACCESS QBE AND ITS SQL



Numerous conditional restrictions can be placed on the selected table contents. For example, the comparison operators shown in Table 7.6 can be used to restrict output. Note that there are two options for *not equal to*. Both <> and != are well supported and perform the same function.

TABLE 7.6
COMPARISON OPERATORS

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

The following example uses one of the *not equal to* operators:

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

The output, shown in Figure 7.23, lists all of the rows for which the vendor code is *not* 21344.

FIGURE 7.23 PRODUCT ATTRIBUTES FOR VENDOR CODES OTHER THAN 21344

P_DESCRIPT	P_QOH	P_PRICE	V_CODE
Power painter, 15 psi., 3-nozzle	8	109.99	25595
Hrd. cloth, 1/4-in., 2x50	15	39.95	23119
Hrd. cloth, 1/2-in., 3x50	23	43.99	23119
B&D jigsaw, 12-in. blade	8	109.92	24288
B&D jigsaw, 8-in. blade	6	99.87	24288
B&D cordless drill, 1/2-in.	12	39.95	25595
Claw hammer	23	9.95	21225
Hicut chain saw, 16 in.	11	256.99	24288
1.25-in. metal screw, 25	172	6.99	21225
2.5-in. wd. screw, 50	237	8.45	21231
Steel matting, 4'x8'x1/6", .5" mesh	18	119.95	25595

Note that, in Figure 7.23, rows with nulls in the V_CODE column (see Figure 7.2) are not included in the SELECT command's output.

The following command sequence:

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

yields the output shown in Figure 7.24.

FIGURE 7.24 SELECT PRODUCT TABLE ATTRIBUTES WITH A P_PRICE RESTRICTION

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

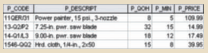
7-6b Using Comparison Operators on Character Attributes

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

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

would be correct and would yield a list of all rows in which the P_CODE is alphabetically less than 1558-QW1. (Because the ASCII code value for the letter B is greater than the value of the letter A, it follows that A is less than B.) Therefore, the output will be generated as shown in Figure 7.25.

FIGURE 7.25 THE ASCII CODE EFFECT



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

String (character) comparisons are made from left to right. This left-to-right comparison is especially useful when attributes such as names are to be compared. For example, the string "Ardmore" would be judged *greater than* the string "Aarenson" but *less than* the string "Brown"; such results may be used to generate alphabetical listings like those in a phone directory. If the characters 0–9 are stored as strings, the same left-to-right string comparisons can lead to apparent anomalies. For example, the ASCII code for the character "5" is *greater than* the ASCII code for the character "4," as expected. Yet, the same "5" will also be judged *greater than* the string "44" because the *first* character in the string "44" is less than the string "5."

Due to left-to-right string comparisons, you may get some unexpected results from comparisons when dates or other numbers are stored in character format. For example, the left-to-right ASCII character comparison would force the conclusion that the date "01/01/2018" occurred *before* "12/31/2017." Because the leftmost character "0" in "01/01/2018" is *less than* the leftmost character "1" in "12/31/2017," "01/01/2018" is *less than* "12/31/2017." Naturally, if date strings are stored in a yyyy/mm/dd format, the comparisons will yield appropriate results, but this is a nonstandard date presentation. Therefore, all current RDBMSs support date data types; you should use them. In addition, using date data types gives you the benefit of date arithmetic.

7-6c Using Comparison Operators on Dates

Date procedures are often more software-specific than other SQL procedures. For example, the query to list all of the rows in which the inventory stock dates occur on or after January 20, 2018, looks like this:

```
SELECT      P_DESCRIPT, P_QOH, P_MIN, P_PRICE, P_INDATE
FROM        PRODUCT
WHERE       P_INDATE >= '20-Jan-2018';
```

Remember that MS Access users must use the # delimiters for dates. For example, you would use #20-Jan-18# in the preceding WHERE clause. The date-restricted output is shown in Figure 7.26. In MySQL, the expected date format is yyyy-mm-dd, so the WHERE clause would be written as:

```
WHERE       P_INDATE >= '2018-01-20'
```


FIGURE 7.26 SELECTED PRODUCT TABLE ATTRIBUTES: DATE RESTRICTION

P_DESCRIPT	P_QOH	P_MIN	P_PRICE	P_INDATE
B&D cordless drill, 1/2-in.	12	5	38.95	20-Jan-18
Claw hammer	23	10	9.95	20-Jan-18
Hicut chain saw, 16 in.	11	5	256.99	07-Feb-18
PVC pipe, 3.5-in., 8-ft	188	75	5.87	20-Feb-18
1.25-in. metal screw, 25	172	75	6.99	01-Mar-18
2.5-in. wd. screw, 50	237	100	8.45	24-Feb-18

7-6d Logical Operators: AND, OR, and NOT

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

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

This command generates the six rows shown in Figure 7.27 that match the logical restriction.

FIGURE 7.27 THE LOGICAL OR

P_DESCRIPT	P_QOH	P_PRICE	V_CODE
7.25-in. pwr. saw blade	32	14.99	21344
9.00-in. pwr. saw blade	18	17.49	21344
B&D jigsaw, 12-in. blade	8	109.92	24288
B&D jigsaw, 8-in. blade	6	99.87	24288
Rat-tail file, 1/8-in. fine	43	4.99	21344
Hicut chain saw, 16 in.	11	256.99	24288

The logical operator **AND** has the same SQL syntax requirement as OR. The following command generates a list of all rows for which P_PRICE is greater than $100 *and* for which P_QOH is less than 20:

```
SELECT      P_DESCRIPT, P_QOH, P_PRICE, V_CODE
FROM        PRODUCT
WHERE       P_PRICE > 100
AND         P_QOH < 20;
```

This command produces the output shown in Figure 7.28.

OR
The SQL logical operator used to link multiple conditional expressions in a WHERE or HAVING clause. It requires only one of the conditional expressions to be true.

AND
The SQL logical operator used to link multiple conditional expressions in a WHERE or HAVING clause. It requires that all conditional expressions evaluate to true.

FIGURE 7.28 THE LOGICAL AND

P_DESCRIPT	P_QOH	P_PRICE	V_CODE
Power painter, 15 psi., 3-nozzle	8	109.99	25595
B&D jigsaw, 12-in. blade	8	109.92	24288
Hicut chain saw, 16 in.	11	256.99	24288
Steel matting, 4'x8'x1/6", .5" mesh	18	119.95	25595

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

- The V_CODE is either 25595 *or* 24288.
- *And* the P_PRICE is greater than $100.

The following code produces incorrect results. As shown in Figure 7.29, all rows from vendor 25595 are included in the result even though some of the P_PRICE values are less than the required $100. This is because the DBMS executes the AND operator before the OR operator.

```
SELECT      P_DESCRIPT, P_PRICE, V_CODE
FROM        PRODUCT
WHERE       V_CODE = 25595 OR V_CODE = 24288 AND P_PRICE > 100;
```

FIGURE 7.29 INCORRECT COMBINATION OF AND AND OR

P_DESCRIPT	P_PRICE	V_CODE
Power painter, 15 psi., 3-nozzle	109.99	25595
B&D jigsaw, 12-in. blade	109.92	24288
B&D cordless drill, 1/2-in.	38.95	25595
Hicut chain saw, 16 in.	256.99	24288
Steel matting, 4'x8'x1/6", .5" mesh	119.95	25595

The conditions in the WHERE clause can be grouped using parentheses to produce the desired result. The required listing can be produced by using the following:

```
SELECT      P_DESCRIPT, P_PRICE, V_CODE
FROM        PRODUCT
WHERE       (V_CODE = 25595 OR V_CODE = 24288) AND P_PRICE > 100;
```

Note the use of parentheses to combine logical restrictions. Where you place the parentheses depends on how you want the logical restrictions to be executed. Conditions listed within parentheses are always executed first. The preceding query yields the output shown in Figure 7.30.

FIGURE 7.30 CORRECT COMBINATION AND AND OR CONDITIONS

P_DESCRIPT	P_PRICE	V_CODE
Power painter, 15 psi., 3-nozzle	109.99	25595
B&D jigsaw, 12-in. blade	109.92	24288
Hicut chain saw, 16 in.	256.99	24288
Steel matting, 4'x8'x1/6", .5" mesh	119.95	25595

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

The logical operator **NOT** is used to negate the result of a conditional expression. That is, in SQL, all conditional expressions evaluate to true or false. If an expression is true, the row is selected; if an expression is false, the row is not selected. The NOT logical operator is typically used to find the rows that *do not* match a certain condition. For example, if you want to see a listing of all rows for which the vendor code is *not* 21344, use the following command sequence:

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

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

7-6e Old-Style Joins

In Chapter 3, you learned that a natural join can conceptually be thought of as a three-step process: (1) create a product between the tables, (2) use the relational selection operation to restrict to only the rows that have matching values for the common attribute, and (3) use relational projection to drop a copy of the common attribute. An equijoin was then shown to be the result of performing just the first two of those three steps. Although best practices discourage performing a join using these literal steps, it is still possible to do. For example, you can join the PRODUCT and VENDOR tables through their common V_CODE by writing the following:

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

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

Note the following points about the preceding query:

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

Boolean algebra
A branch of mathematics that uses the logical operators OR, AND, and NOT.

NOT
A SQL logical operator that negates a given predicate.

Old-style joins are generally not recommended because of two potential problems. First, the task of joining the tables is split across both the FROM and WHERE clauses, which makes complex queries more difficult to maintain. Having a clear separation of responsibilities among the SELECT query clauses makes code maintenance easier. With JOIN ON or JOIN USING syntax, all of the code necessary to join the tables together is located in the FROM clause. All of the code necessary to restrict the data based on business requirements is located in the WHERE clause. With an old-style join, the criteria for completing the join are mixed with the criteria to restrict the data based on business requirements. Second, the old-style join is susceptible to undetected errors that other joins are not. For example, the following query attempts to join multiple tables to list the customers that have purchased products that are supplied by vendors from TN, but it contains an error. The join condition to link the LINE table and the PRODUCT table is missing. As a result, the query generates an error.

```
SELECT    CUS_FNAME, CUS_LNAME, V_NAME
FROM      CUSTOMER JOIN INVOICE ON CUSTOMER.CUS_CODE =
          INVOICE.CUS_CODE JOIN LINE ON INVOICE.INV_NUMBER =
          LINE.INV_NUMBER JOIN PRODUCT JOIN VENDOR ON
          PRODUCT.V_CODE = VENDOR.V_CODE
WHERE     V_STATE = 'TN';
```

In the previous query, the DBMS can detect that there is a missing join condition because every JOIN must have a join condition.

The following query, using old-style joins, contains the exact same error. However, it does not generate an error from the DBMS—it simply provides the users with incorrect data! The DBMS cannot relate the intended joins with the criteria in the WHERE clause, so it cannot detect the missing join condition.

```
SELECT    CUS_FNAME, CUS_LNAME, V_NAME
FROM      CUSTOMER, INVOICE, LINE, PRODUCT, VENDOR
WHERE     V_STATE = 'TN' AND CUSTOMER.CUS_CODE =
          INVOICE.CUS_CODE AND INVOICE.INV_NUMBER =
          LINE.INV_NUMBER AND PRODUCT.V_CODE =
          VENDOR.V_CODE;
```

7-6f Special Operators

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

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

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

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

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

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

BETWEEN
In SQL, a special comparison operator used to check whether a value is within a range of specified values.

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


Note

When using the BETWEEN special operator, always specify the lower-range value first. The WHERE clause of the command above is interpreted as:

```
WHERE P_PRICE >= 50 AND P_PRICE <= 100
```

If you list the higher-range value first, the DBMS will return an empty result set because the WHERE clause will be interpreted as:

```
WHERE P_PRICE >= 100 AND P_PRICE <= 50
```

Clearly, no product can have a price that is both greater than 100 and simultaneously less than 50. Therefore, no rows can possibly match the criteria.

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

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

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

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

can be handled more efficiently with:

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

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

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

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

IN
In SQL, a comparison operator used to check whether a value is among a list of specified values.

LIKE
In SQL, a comparison operator used to check whether an attribute's text value matches a specified string pattern.

The IN operator is especially valuable when it is used in conjunction with subqueries, which we will discuss in a later section.

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

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

Note

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

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

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

Figure 7.31 shows that the results include contacts named "Smith" and "Smithson."

FIGURE 7.31 VENDOR CONTACTS THAT START WITH "SMITH"

V_NAME	V_CONTACT	V_AREACODE	V_PHONE
Bryson, Inc.	Smithson	615	223-3234
Dome Supply	Smith	901	678-1419
B&K, Inc.	Smith	904	227-0093

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

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

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

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

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

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

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

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

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

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

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

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

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

The IS NULL Special Operator Standard SQL allows the use of **IS NULL** to check for a null attribute value. For example, suppose that you want to list all products that do not have a vendor assigned (i.e., the V_CODE attribute does not contain a value). Such a null entry could be found by using the following command sequence, as shown in Figure 7.32.

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

FIGURE 7.32 PRODUCTS NOT ASSOCIATED WITH A VENDOR

P_CODE	P_DESCRIPT	V_CODE
23114-AA	Sledge hammer, 12 lb.	
PVC23DRT	PVC pipe, 3.5-in., 8-ft	

IS NULL
In SQL, a comparison operator used to check whether an attribute has a value.

Note that SQL uses a special operator to test for nulls. Why? Couldn't you just enter a condition such as "V_CODE = NULL"? No. Technically, NULL is not a "value" the way the number 0 or the blank space is; instead, a NULL is a special property of an attribute that represents the absence of any value. For logical comparisons, NULL can be thought of as being Unknown. If V_CODE = NULL is used in a WHERE clause, when the value 21225 for V_CODE is evaluated, the DBMS considers, "Is 21225 equal to Unknown?" The answer is unknown because the DBMS does not know what the value Unknown is supposed to represent. When a NULL in V_CODE is evaluated for this condition, the DBMS considers, "Is Unknown equal to Unknown?" The answer is unknown because the DBMS does not know what value the first Unknown represents nor the value of the second Unknown, so it cannot say if they represent the same or different values. For the WHERE clause to include a row in the resulting query, the criteria must evaluate to True, so False or Unknown results are not included. Therefore, WHERE V_CODE = NULL will never return any rows since every row, regardless of whether or not it contains a value for V_CODE, will evaluate to Unknown. Thus, the IS NULL operator is used instead.

The use of the IS NULL operator is useful in many situations. For example, it is often used to find unmatched rows, as shown in Figure 7.32. Finding unmatched rows on the "many" side of a 1:M relationship is simplified because the foreign key within the table contains nulls. Finding unmatched rows on the "one" side of a 1:M can be done by using the IS NULL operator in conjunction with an outer join, such as finding the vendors that are not associated with any product as shown in Figure 7.33.

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

FIGURE 7.33 VENDORS NOT ASSOCIATED WITH ANY PRODUCT

V_CODE	V_NAME	P_CODE
21226	SuperLoo, Inc.	
22567	Dome Supply	
24004	Brackman Bros.	
25443	B&K, Inc.	
25501	Damal Supplies	

Special Operators with NOT As discussed previously, the NOT logical connector can be used to negate a condition. NOT can also be used with the special operators as well. While BETWEEN will return rows with values within a given range of values, NOT BETWEEN will return rows with values that are outside of the given range. IN will return rows with values that match any value within a given list. NOT IN will return rows with values that do not match any value within the given list. NOT IN can be a little trickier than it sounds at first. For NOT IN to return a row, the value being compared against the list must evaluate to False when compared against every value in the list. Remember, NULL is the absence of a value. When a value is logically compared against NULL, it does not evaluate as True or False, it evaluates to Unknown. Therefore, if the list of values used with NOT IN contains a null, then the operator will not return any rows. LIKE is used for a substring search to find a smaller string of text within a larger string of text. NOT LIKE will return the rows that do not contain the smaller string of text. IS NULL

returns rows that do not have a value in the specified attribute. Unlike the other special operators that place the word NOT in front of the operator, IS NULL places the word NOT in the middle to produce the IS NOT NULL operator. IS NOT NULL returns rows that contain any value in the specified attribute, regardless of what that value is.

7-7 Aggregate Processing

Consider the following query:

```
SELECT    V_CODE, V_NAME, V_STATE, P_CODE, P_DESCRIPT,
          P_PRICE * P_QOH AS TOTAL
FROM      PRODUCT P JOIN VENDOR V ON P.V_CODE = V.V_CODE
WHERE     V_STATE IN ('TN', 'KY')
ORDER BY  V_STATE, TOTAL DESC;
```

If we consider the processing of this query, it almost *appears* as if the RDBMS is operating one row at a time. Each row in the PRODUCT table is compared against each row in the VENDOR table to find the matching rows. Those matching rows are then filtered by looking at each row to see if the vendor state matches a value in the list. For each filtered row, the specified columns are retrieved and the computed field is calculated. Using the columns returned, the rows are evaluated by state and total to sort the rows to produce the final output shown in Figure 7.34. The RDBMS actually works on sets of data, not individual row processing, but one can imagine the processing that has been discussed up to this point as if it were row-based.

FIGURE 7.34 TOTAL VALUE OF PRODUCTS FROM VENDOR IN TN OR KY

V_CODE	V_NAME	V_STATE	P_CODE	P_DESCRIPT	TOTAL
21344	Gomez Bros.	KY	13-Q2/P2	7.25-in. pwr. saw blade	479.68
21344	Gomez Bros.	KY	14-Q1/L3	9.00-in. pwr. saw blade	314.82
21344	Gomez Bros.	KY	54778-2T	Rat-tail file, 1/8-in. fine	214.57
24288	ORDVA, Inc.	TN	89-WRE-Q	Hicut chain saw, 16 in.	2826.89
21231	D&E Supply	TN	SVV-23116	2.5-in. wd. screw, 50	2002.65
21225	Bryson, Inc.	TN	SM-18277	1.25-in. metal screw, 25	1202.28
24288	ORDVA, Inc.	TN	2232/QTY	B&D jigsaw, 12-in. blade	879.36
24288	ORDVA, Inc.	TN	2232/QWE	B&D jigsaw, 8-in. blade	599.22
21225	Bryson, Inc.	TN	23109-HB	Claw hammer	228.85

However, there are many questions that are asked of the database that require working with collections of rows as if they are a single unit. This type of collection processing is done with aggregate functions. The defining characteristic of using an aggregate function is that it takes a collection of rows and reduces it to a single row. SQL provides useful aggregate functions that count, find minimum and maximum values, calculate averages, and so on. Better yet, SQL allows the user to limit queries to only those entries that have no duplicates or entries whose duplicates can be grouped.

7-7a Aggregate Functions

SQL can perform various mathematical summaries for you, such as counting the number of rows that contain a specified condition, finding the minimum or maximum values for a specified attribute, summing the values in a specified column, and averaging

the values in a specified column. While there are other aggregate functions supported by some DBMS products, the ones shown in Table 7.7 are the most common aggregate functions and are supported by most DBMS products. Aggregate functions are most commonly used in the SELECT column list to return an aggregate value that has been calculated across a collection of rows.

TABLE 7.7
SOME BASIC SQL AGGREGATE FUNCTIONS

FUNCTION	OUTPUT
COUNT	The number of rows containing non-null values
MIN	The minimum attribute value encountered in a given column
MAX	The maximum attribute value encountered in a given column
SUM	The sum of all values for a given column
AVG	The arithmetic mean (average) for a specified column

COUNT The **COUNT** function is used to tally the number of non-null values of an attribute. In the following code, a tally of the number of products is calculated, returning a result of 16 (see Figure 7.35).

```
SELECT    COUNT (P_CODE)
FROM      PRODUCT;
```

FIGURE 7.35 COUNT OF PRODUCT CODES IN THE PRODUCT TABLE

CountOfP_CODE
16

Notice that the aggregate function took the entire collection of rows from the PRODUCT table and reduced it to a single row for the result. This is one of the defining behaviors of aggregates—reduce collections of rows to a single row. The collection of rows does not have to be composed of all of the rows in the table. For example, we can use the following query to determine how many products have a price that is less than $10.

```
SELECT    COUNT(P_PRICE)
FROM      PRODUCT
WHERE     P_PRICE < 10;
```

COUNT
A SQL aggregate function that outputs the number of rows containing not null values for a given column or expression, sometimes used in conjunction with the DISTINCT clause.

Aggregate functions take a value, typically an attribute, as a parameter inside parentheses. In Figure 7.35, the code counted the values in the primary key, P_CODE, in the PRODUCT table. Notice the difference in the result when counting the V_CODE attribute in that same table, as shown in Figure 7.36.

```
SELECT    COUNT(V_CODE)
FROM      PRODUCT;
```


FIGURE 7.36 COUNT OF VENDOR CODES IN THE PRODUCT TABLE

CountOfV_CODE
14

In this case, the V_CODE attribute contains nulls. COUNT does not include nulls in the tally. If the purpose of a query is to return the number of rows in a collection, regardless of whether or not any specific column contains nulls, then the syntax COUNT(*) can be used. Most aggregate functions take a single attribute as the parameter; however, COUNT allows the asterisk (*) wildcard to indicate that the number of rows should be returned without regard to the values in any particular column.

COUNT can also be used in conjunction with the DISTINCT clause. The previous example of using the DISTINCT keyword earlier in this chapter showed the DISTINCT immediately following the SELECT keyword to eliminate duplicate rows in the query result, as shown in Figure 7.7. In Figure 7.7, a list of the different vendor codes was returned. However, suppose that you want to find out *how many* different vendors are in the PRODUCT table. Instead of placing DISTINCT immediately after the SELECT keyword, the DISTINCT can be placed inside the COUNT function (see Figure 7.37).

```
SELECT    COUNT(DISTINCT V_CODE) AS "COUNT DISTINCT"
FROM      PRODUCT;
```

FIGURE 7.37 COUNT OF DISTINCT VENDOR CODES

Count Distinct
6

In this case, the DISTINCT will be applied to the values in the attribute before the tally is calculated by the COUNT. Note that the nulls are not counted as V_CODE values. The FROM clause retrieves all of the values from the PRODUCT table, the DISTINCT removes the duplicate values in V_CODE, and then COUNT tallies the non-null values returned by the DISTINCT.

Note

MS Access does not support the use of the COUNT with the DISTINCT clause. If you want to use such queries in MS Access, you must create subqueries (discussed later in this chapter) with DISTINCT and NOT NULL clauses. For example, the equivalent MS Access queries for the two queries above are:

```
SELECT  COUNT(*)
FROM    (SELECT DISTINCT V_CODE FROM PRODUCT WHERE V_CODE IS NOT NULL);
```

and

```
SELECT  COUNT(*)
FROM    (SELECT DISTINCT V_CODE
         FROM (SELECT V_CODE, P_PRICE FROM PRODUCT
                  WHERE V_CODE IS NOT NULL AND P_PRICE < 10));
```

Subqueries are discussed in detail later in this chapter.

MIN and MAX The **MIN** and **MAX** functions help you find answers to problems such as the highest and lowest (maximum and minimum) prices in the PRODUCT table. The examples of the COUNT function illustrated that aggregate functions reduce a collection of rows into a single row. Retrieving a single column, however, is not required. The previous examples returned a single column because only one column was specified in the SELECT column list of the query. The following code retrieves the highest and lowest prices in the PRODUCT table in a single query (see Figure 7.38).

```
SELECT    MAX(P_PRICE) AS MAXPRICE, MIN(P_PRICE) AS MINPRICE
FROM      PRODUCT;
```

FIGURE 7.38 MAXIMUM AND MINIMUM PRICE OUTPUT

MAXPRICE	MINPRICE
256.99	4.99

The MAX and MIN aggregate functions can also be used with date columns. Recall from the earlier explanation of date arithmetic that dates are stored in the database as a day number, that is, the number of days that have passed since some defined point in history. As a day number, yesterday is one less than today, and tomorrow is one more than today. Therefore, older dates are "smaller" than future dates, so the oldest date would be the smallest date and the most future date would be the largest date. For example, to find out which product has the oldest inventory date, you would use MIN(P_INDATE). In the same manner, to find out the most recent inventory date for a product, you would use MAX(P_INDATE).

SUM and AVG The **SUM** function computes the total sum for any specified numeric attribute, using any condition(s) you have imposed. For example, if you want to compute the total amount owed by your customers, you could use the following command:

```
SELECT    SUM(CUS_BALANCE) AS TOTBALANCE
FROM      CUSTOMER;
```

An aggregate function takes a value as a parameter, such as CUS_BALANCE in the previous query. The value is typically an attribute stored in a table. However, derived attributes and formulas are also acceptable. For example, if you want to find the total value of all items carried in inventory, you could use the following:

```
SELECT    SUM(P_QOH * P_PRICE) AS TOTVALUE
FROM      PRODUCT;
```

The total value is the sum of the product of the quantity on hand and the price for all items. (See Figure 7.39.)

FIGURE 7.39 TOTAL VALUE OF ALL ITEMS IN THE PRODUCT TABLE

TOTVALUE
15084.52

MIN
A SQL aggregate function that yields the minimum attribute value in a given column.

MAX
A SQL aggregate function that yields the maximum attribute value in a given column.

SUM
A SQL aggregate function that yields the sum of all values for a given column or expression.

The **AVG** function format is similar to those of SUM and is subject to the same operating restrictions. The following command set shows how a simple average P_PRICE value can be generated to yield the computed average price of 56.42125, as shown in Figure 7.40.

```
SELECT    AVG(P_PRICE) AS AVGPRICE
FROM      PRODUCT;
```

FIGURE 7.40 AVERAGE PRODUCT PRICE

AVGPRICE
56.42125

7-7b Grouping Data

In the previous examples, the aggregate functions summarized data across all rows in the given tables. Sometimes, however, you do not want to treat the entire table as a single collection of data for summarizing. Rows can be grouped into smaller collections quickly and easily using the **GROUP BY** clause within the SELECT statement. The aggregate functions will then summarize the data within each smaller collection. The syntax is:

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

Figure 7.40 determined the average price of all of the products in the database. However, what if instead of seeing the price across all products, the users wanted to see the average price of the products provided by each vendor? The following query will answer that question, as shown in Figure 7.41.

```
SELECT    V_CODE, AVG(P_PRICE) AS AVGPRICE
FROM      PRODUCT
GROUP BY  V_CODE;
```

FIGURE 7.41 AVERAGE PRICE OF PRODUCTS FROM EACH VENDOR

V_CODE	AVGPRICE
	10.13
21225	8.47
21231	8.45
21344	12.49
23119	41.97
24288	155.59
25595	89.63

AVG
A SQL aggregate function that outputs the mean average for a specified column or expression.

GROUP BY
A SQL clause used to create frequency distributions when combined with any of the aggregate functions in a SELECT statement.

Instead of treating all of the PRODUCT rows as a single collection, the query separates the rows into several smaller collections, each collection based on the value of V_CODE. Therefore, all of the products from vendor 21225 are placed into one collection, all of the products from vendor 21344 are placed into a second collection, all of the products from vendor 25595 are placed in a third collection, and so on until all of the products appear in a collection. These collections are formed using the GROUP BY clause. GROUP BY forms the collections based on the value of V_CODE, and then the aggregate function reduces each collection to a single row and calculates the average price for that collection. The aggregate function still does what aggregates always do—reduce a collection of rows to a single row—but in this case there are multiple collections. Note that the products with null vendor codes in Figure 7.41 are grouped together. Aggregate functions ignore nulls when performing calculations, but the GROUP BY clause includes nulls and considers all of the nulls to be the same when forming collections.

Understanding the interaction between the GROUP BY clause and aggregate functions is crucial in using them correctly. Consider the following query, whose result is shown in Figure 7.42:

```
SELECT      V_CODE, V_NAME, COUNT(P_CODE) AS NUMPROD,
            AVG(P_PRICE) AS AVGPRICE
FROM        PRODUCT JOIN VENDOR ON PRODUCT.V_CODE =
            VENDOR.V_CODE
GROUP BY    V_CODE, V_NAME
ORDER BY    V_NAME;
```

FIGURE 7.42 COUNT OF PRODUCTS AND AVERAGE PRICES FROM EACH VENDOR

V_CODE	V_NAME	NUMPRODS	AVGPRICE
21225	Bryson, Inc.	2	8.47
21231	D&E Supply	1	8.45
21344	Gomez Bros.	3	12.49
23119	Randsets Ltd.	2	41.97
24288	ORDVA, Inc.	3	155.59
25595	Rubicon Systems	3	89.63

In this query, first the DBMS retrieves the data from the PRODUCT and VENDOR tables and joins them using V_CODE as the common attribute. Next, the resulting rows are grouped into collections based on rows on having the same values for both V_CODE and V_NAME. Third, the SELECT column list projects out just the V_CODE, V_NAME, P_CODE, and P_PRICE attributes. The aggregate functions then reduce each collection to a single row. Within a collection, the prices are averaged, and the product codes are counted. Because each collection was formed based on having the same value for vendor code and vendor name, the DBMS knows with certainty that every row in a collection has the same value for those attributes. Therefore, when the collection is reduced to one row, the DBMS knows that it can display the V_CODE and V_NAME for that collection because all rows in the collection have the same values. Finally, the ORDER BY clause sorts the resulting rows for each collection in ascending order by vendor name.

Now, consider the same query but with one additional attribute added to the SELECT column list:

```
SELECT      V_CODE, V_NAME, P_QOH, COUNT(P_CODE), AVG(P_PRICE)
FROM        PRODUCT JOIN VENDOR ON PRODUCT.V_CODE =
            VENDOR.V_CODE
GROUP BY    V_CODE, V_NAME
ORDER BY    V_NAME;
```

This query will not execute but generates a "not a GROUP BY expression" error. The FROM clause operates exactly the same as in the previous query to join the PRODUCT and VENDOR tables. The GROUP BY clause operates exactly the same as in the previous query to form collections of the rows based on vendor code and vendor name. The SELECT column list projects the product quantity on hand in addition to the vendor code, vendor name, product code, and product price. The error occurs when the aggregate functions attempt to reduce each collection to a single row. The aggregates can reduce price by taking an average, reduce product code by counting, display vendor code and vendor name because all rows in the collection have the same value, but what about the quantity on hand attribute? Each row in a collection may have different values for the quantity on hand for the products in that collection. The query does not group by P_QOH, so the DBMS cannot know with certainty that the rows all have the same value. The query does not apply an aggregate function to the P_QOH attribute, so the DBMS does not know how to calculate a single value to represent P_QOH in the collection. Therefore, an error is generated. To fix this error, either an aggregate function must be applied to P_QOH so that a single value can be calculated, or P_QOH must be added to the GROUP BY clause so that the DBMS can enforce that every row in the group has the same value for that attribute. Notice the difference in the results of these two possible solutions.

```
SELECT      V_CODE, V_NAME, SUM(P_QOH) AS TOTALQTY,
            COUNT(P_CODE) AS NUMPRODS, AVG(P_PRICE) AS AVGPRICE
FROM        PRODUCT JOIN VENDOR ON PRODUCT.V_CODE =
            VENDOR.V_CODE
GROUP BY    V_CODE, V_NAME
ORDER BY    V_NAME;
```

Figure 7.43 shows the result with a SUM function being applied to P_QOH. The result has six rows.

FIGURE 7.43 GROUPING BY VENDOR CODE AND VENDOR NAME

V_CODE	V_NAME	TOTALQTY	NUMPRODS	AVGPRICE
21225	Bryson, Inc.	195	2	8.47
21231	D&E Supply	237	1	8.45
21344	Gomez Bros.	93	3	12.49
23119	Randsets Ltd.	38	2	41.97
24288	ORDVA, Inc.	25	3	155.59
25595	Rubicon Systems	38	3	89.63

```
SELECT      V_CODE, V_NAME, P_QOH, COUNT(P_CODE) AS NUMPRODS,
            AVG(P_PRICE) AS AVGPRICE
FROM        PRODUCT JOIN VENDOR ON PRODUCT.V_CODE =
            VENDOR.V_CODE
GROUP BY    V_CODE, V_NAME, P_QOH
ORDER BY    V_NAME;
```

Figure 7.44 shows the result when P_QOH is added to the GROUP BY clause. The result has 14 rows and the values of the COUNT and AVG functions have changed since the collections formed by the GROUP BY clause no longer contain the same sets of rows. The number of groups changed because when the rows were grouped by vendor code and vendor name (as in Figure 7.43), there was variation among the rows for the values in P_QOH. In Figure 7.44, collections were formed by requiring all of the rows in a group to have the same value for V_CODE, V_NAME, and P_QOH.

FIGURE 7.44 GROUPS CHANGED BY INCLUSION OF P_QOH

V_CODE	V_NAME	P_QOH	NUMPRODS	AVGPRICE
21225	Bryson, Inc.	23	1	9.95
21225	Bryson, Inc.	172	1	6.99
21231	D&E Supply	237	1	8.45
21344	Gomez Bros.	18	1	17.49
21344	Gomez Bros.	32	1	14.99
21344	Gomez Bros.	43	1	4.99
23119	Randsets Ltd.	15	1	39.95
23119	Randsets Ltd.	23	1	43.99
24288	ORDVA, Inc.	6	1	99.87
24288	ORDVA, Inc.	8	1	109.92
24288	ORDVA, Inc.	11	1	256.99
25595	Rubicon Systems	8	1	109.99
25595	Rubicon Systems	12	1	38.95
25595	Rubicon Systems	18	1	119.95

Including additional attributes in the GROUP BY clause does not always cause the number of groups formed to change. For example, instead of adding P_QOH to the query from Figure 7.42, consider the addition of V_STATE instead. In this case, adding V_STATE to the GROUP BY clause does not change the values for the count of products or the average price because every collection formed from grouping on V_CODE and V_NAME had only one value for V_STATE within that collection, as shown in Figure 7.45.

As you can see, great care must be taken when constructing queries that use groups and aggregate functions because the addition of even one attribute can significantly change the results returned by the query.

7-7c HAVING Clause

Aggregate functions are very powerful and are used frequently in reporting. Most often, aggregate functions appear in the SELECT column list of a query. It is also possible to

FIGURE 7.45 GROUPS INCLUDING V_STATE

V_CODE	V_NAME	V_STATE	NUMPRODS	AVGPRICE
21225	Bryson, Inc.	TN	2	8.47
21231	D&E Supply	TN	1	8.45
21344	Gomez Bros.	KY	3	12.49
23119	Randsets Ltd.	GA	2	41.97
24288	ORDVA, Inc.	TN	3	155.59
25595	Rubicon Systems	FL	3	89.63

use aggregate functions in the ORDER BY clause to sort results based on a calculated aggregate value. However, restricting data based on an aggregate value is slightly more complicated and can require the use of a **HAVING** clause. The syntax for a HAVING clause is:

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

The HAVING clause operates very much like the WHERE clause in the SELECT statement. However, the WHERE clause applies to columns and expressions for individual rows, whereas the HAVING clause is applied to the output of a GROUP BY operation. For example, suppose that you want to generate a listing of the number of products in the inventory supplied by each vendor. However, this time you want to limit the listing to products whose prices average less than $10. The query requires both a GROUP BY clause and a HAVING clause, as illustrated in Figure 7.46.

```
SELECT      V_CODE, COUNT(P_CODE) AS NUMPRODS
FROM        PRODUCT
GROUP BY    V_CODE
HAVING      AVG(P_PRICE) < 10
ORDER BY    V_CODE;
```

FIGURE 7.46 APPLICATION OF THE HAVING CLAUSE

V_CODE	NUMPRODS
21225	2
21231	1

If you use the WHERE clause instead of the HAVING clause, the query in Figure 7.46 will produce an error message. That is not to say that a query cannot contain both a WHERE clause and a HAVING clause, just that the clauses do different things. WHERE

HAVING
A clause applied to the output of a GROUP BY operation to restrict selected rows.

is used to restrict rows and is executed prior to the GROUP BY clause. Because WHERE executes before GROUP BY, WHERE cannot contain an aggregate function because the collections needed by the aggregate function do not exist yet. The HAVING clause is used to restrict groups and is executed after the GROUP BY clause. HAVING can contain aggregate functions because the collections are formed by the GROUP BY clause before the HAVING is executed. HAVING restricts *groups*. It is not possible for a HAVING clause to restrict some rows in a group but leave others. HAVING either keeps or eliminates the entire group, so the condition in the HAVING clause must be applicable to the entire group. For this reason, HAVING clauses not only are allowed to contain aggregate functions, but they almost always do.

You can combine multiple clauses and aggregate functions. For example, consider the following SQL statement:

```
SELECT      V_CODE, V_NAME, SUM(P_QOH * P_PRICE) AS TOTCOST
FROM        PRODUCT JOIN VENDOR ON PRODUCT.V_CODE =
            VENDOR.V_CODE
WHERE       P_DISCOUNT > 0
GROUP BY    V_CODE, V_NAME
HAVING      (SUM(P_QOH * P_PRICE) > 500)
ORDER BY    SUM(P_QOH * P_PRICE) DESC;
```

This statement does the following:

- Joins the product and vendor tables using V_CODE as the common attribute
- Restricts to only the rows with a discount greater than 0
- Groups the remaining rows into collections based on V_CODE and V_NAME
- Aggregates the total cost of products in each group
- Restricts to only the groups with totals that exceed $500
- Lists the results in descending order by the total cost

Note the syntax used in the HAVING and ORDER BY clauses; in both cases, you should specify the column expression (formula) used in the SELECT statement's column list, rather than the column alias (TOTCOST). Some RDBMSs allow you to replace the column expression with the column alias, while others do not.

7-8 Subqueries

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

```
SELECT      INV_NUMBER, INVOICE.CUS_CODE, CUS_LNAME,
            CUS_FNAME
FROM        CUSTOMER C JOIN INVOICE I ON C.CUS_CODE = I.CUS_CODE;
```

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

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


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

However, this result can also be found by using a **subquery**, such as:

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

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

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

In both queries, you needed to get information that was not previously known:

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

In both cases, you used a subquery to generate the required information, which could then be used as input for the originating query. There are key characteristics that you should remember for subqueries:

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

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

The subquery is always on the right side of a comparison or assigning expression. Also, a subquery can return one or more values. To be precise, the subquery can return the following:

- *One single value (one column and one row)*. This subquery is used anywhere a single value is expected, as in the right side of a comparison expression. An example is the preceding query, in which you retrieved products with a price greater than the average price of products.
- *A list of values (one column and multiple rows)*. This type of subquery is used anywhere a list of values is expected, such as when using the IN clause—for example, when comparing the vendor code to a list of vendors as above. Again, in this case, there is only one column of data with multiple value instances. This type of subquery

subquery
A query that is embedded (or nested) inside another query. Also known as a *nested query* or an *inner query*.

is used frequently in combination with the IN operator in a WHERE conditional expression.

- *A virtual table (multicolumn, multirow set of values)*. This type of subquery can be used anywhere a table is expected, such as when using the FROM clause. You will see an example later in this chapter.

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

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

7-8a WHERE Subqueries

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

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

The output of the preceding query is shown in Figure 7.47. Note that this type of query, when used in a >, <, =, >=, or <= conditional expression, requires a subquery that returns only one value (one column, one row). The value generated by the subquery must be of a comparable data type; if the attribute to the left of the comparison symbol is a character type, the subquery must return a character string. Also, if the query returns more than a single value, the DBMS will generate an error.

FIGURE 7.47 PRODUCTS WITH A PRICE GREATER THAN THE AVERAGE PRICE

P_CODE	P_PRICE
11QER/31	109.99
2232/QTY	109.92
2232/QWE	99.87
89-WRE-Q	256.99
WR3/TT3	119.95

Note

You can use an expression anywhere a column name is expected. Suppose that you want to know what product has the highest inventory value. To find the answer, you can write the following query:

```
SELECT    *
FROM      PRODUCT
WHERE     P_QOH * P_PRICE = (SELECT MAX(P_QOH * P_PRICE)
                             FROM PRODUCT);
```


Subqueries can also be used in combination with joins. For example, the following query lists all customers who ordered a claw hammer:

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

The result of the query is shown in Figure 7.48.

FIGURE 7.48 CUSTOMERS WHO ORDER A CLAW HAMMER

CUS_CODE	CUS_LNAME	CUS_FNAME
10011	Dunne	Leona
10014	Orlando	Myron

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

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

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

7-8b IN Subqueries

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

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


The result of the query is shown in Figure 7.49.

FIGURE 7.49 IN SUBQUERY EXAMPLE

CUS_CODE	CUS_LNAME	CUS_FNAME
10011	Dunne	Leona
10012	Smith	Kathy
10014	Orlando	Myron
10015	O'Brian	Amy

7-8c HAVING Subqueries

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

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

The result of the query is shown in Figure 7.50.

FIGURE 7.50 HAVING SUBQUERY EXAMPLE

P_CODE	TOTALUNITS
13-Q2/P2	8
23109-HB	5
54778-2T	6
PVC23DRT	17
SM-18277	3
WR3/TT3	3

7-8d Multirow Subquery Operators: ALL and ANY

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

For example, suppose you want to know which products cost more than all individual products provided by vendors from Florida:

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

The result of the query is shown in Figure 7.51.

FIGURE 7.51 MULTIROW SUBQUERY EXAMPLE

P_CODE	TOTALVALUE
89-WRE-Q	2826.89

It is important to note the following points about the query and its output in Figure 7.51:

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

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

7-8e FROM Subqueries

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

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

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

The result of the query is shown in Figure 7.52.

FIGURE 7.52 FROM SUBQUERY EXAMPLE

CUS_CODE	CUS_LNAME
10014	Orlando

Note in Figure 7.52 that the first subquery returns all customers who purchased product 13-Q2/P2, while the second subquery returns all customers who purchased product 23109-HB. So, in this FROM subquery, you are joining the CUSTOMER table with two virtual tables. The join condition selects only the rows with matching CUS_CODE values in each table (base or virtual).

7-8f Attribute List Subqueries

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

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

Figure 7.53 shows the result of the query.

In Figure 7.53, note that the inline query output returns one value (the average product's price) and that the value is the same in every row. Note also that the query uses the

FIGURE 7.53 INLINE SUBQUERY EXAMPLE

P_CODE	P_PRICE	AVGPRICE	DIFF
11QER/31	109.99	56.42125	53.56875
13-Q2/P2	14.99	56.42125	-41.43125
14-Q1/L3	17.49	56.42125	-38.93125
1546-QQ2	39.95	56.42125	-16.47125
1558-QWI	43.99	56.42125	-12.43125
2232/QTY	109.92	56.42125	53.49875
2232/QWE	99.87	56.42125	43.44875
2238/QPD	38.95	56.42125	-17.47125
23109-HB	9.95	56.42125	-46.47125
23114-AA	14.40	56.42125	-42.02125
54778-2T	4.99	56.42125	-51.43125
89-WRE-Q	256.99	56.42125	200.56874
PVC23DRT	5.87	56.42125	-50.55125
SM-18277	6.99	56.42125	-49.43125
SW-23116	8.45	56.42125	-47.97125
WR3/TT3	119.95	56.42125	63.52875

full expression instead of the column aliases when computing the difference. In fact, if you try to use the alias in the difference expression, you will get an error message. The column alias cannot be used in computations in the attribute list when the alias is defined in the same attribute list. That DBMS requirement is the result of the way the DBMS parses and executes queries.

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

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

The result of the query is shown in Figure 7.54.

As you can see in Figure 7.54, the number of employees remains the same for each row in the result set. The use of this type of subquery is limited to certain instances when you need to include data from other tables that is not directly related to a main table or tables in the query. The value will remain the same for each row, like a constant in a programming language. (You will learn another use of inline subqueries later in this chapter when we discuss correlated subqueries.) Note that you cannot use an alias in the attribute list to write the expression that computes the contribution per employee.

FIGURE 7.54 ANOTHER EXAMPLE OF AN INLINE SUBQUERY

P_CODE	SALES	ECOUNT	CONTRIB
13-Q2/P2	119.92	17	7.05
1546-QQ2	39.95	17	2.35
2232/QTY	109.92	17	6.47
2238/QPD	38.95	17	2.29
23109-HB	49.75	17	2.93
54778-2T	29.94	17	1.76
89-WRE-Q	256.99	17	15.12
PVC23DRT	99.79	17	5.87
SM-18277	20.97	17	1.23
WR3/TT3	359.85	17	21.17

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

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

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

7-8g Correlated Subqueries

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

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

correlated subquery
A subquery that executes once for each row in the outer query.

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


will yield the following output:

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

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

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

This process is the opposite of that of the uncorrelated subqueries, as you have already seen. The query is called a *correlated* subquery because the inner query is *related* to the outer query; the inner query references a column of the outer subquery.

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

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

The following correlated query completes the preceding two-step process, with results shown in Figure 7.55.

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

FIGURE 7.55 CORRELATED SUBQUERY IN WHERE CLAUSE

INV_NUMBER	P_CODE	LINE_UNITS
1003	13-Q2/P2	5
1004	54778-2T	3
1004	23109-HB	2
1005	PVC23DRT	12

In Figure 7.55, note that the LINE table is used more than once, so you must use table aliases. In this case, the inner query computes the average units sold of the product that matches the P_CODE of the outer query P_CODE. That is, the inner query runs once, using the first product code found in the outer LINE table, and it returns the average sale for that product. When the number of units sold in the outer LINE row is greater than the average computed, the row is added to the output. Then the inner

query runs again, this time using the second product code found in the outer LINE table. The process repeats until the inner query has run for all rows in the outer LINE table. In this case, the inner query will be repeated as many times as there are rows in the outer query.

To verify the results and to provide an example of how you can combine subqueries, you can add a correlated inline subquery to the previous query (see Figure 7.56).

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

FIGURE 7.56 TWO CORRELATED SUBQUERIES

INV_NUMBER	P_CODE	LINE_UNITS	AVG
1003	13-Q2/P2	5	2.67
1004	54778-2T	3	2.00
1004	23109-HB	2	1.25
1005	PVC23DRT	12	8.50

As you can see, the new query contains a correlated inline subquery that computes the average units sold for each product. You not only get an answer, but you can also verify that the answer is correct.

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

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

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

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

EXISTS
In SQL, a comparison operator that checks whether a subquery returns any rows.

As shown, the EXISTS special operator can be used with uncorrelated subqueries, but it is almost always used with correlated subqueries. For example, suppose that you want to know the names of all customers who have placed an order lately. In that case, you could use a correlated subquery like the first one shown in Figure 7.57.


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

FIGURE 7.57 CORRELATED SUBQUERY WITH THE EXISTS OPERATOR

CUS_CODE	CUS_LNAME	CUS_FNAME
10011	Dunne	Leona
10012	Smith	Kathy
10014	Orlando	Myron
10015	O'Brian	Amy
10018	Farriss	Anne

Suppose that you want to know which vendors you must contact to order products that are approaching the minimum quantity-on-hand value. In particular, you want to know the vendor code and vendor name for products with a quantity on hand that is less than double the minimum quantity. The query that answers the question is as follows (see Figure 7.58).

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

FIGURE 7.58 VENDORS TO CONTACT

V_CODE	V_NAME
21344	Gomez Bros.
23119	Randsets Ltd.
24288	ORDVA, Inc.
25595	Rubicon Systems

In Figure 7.58, note that:

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

7-9 SQL Functions

The data in databases is the basis of critical business information. Generating information from data often requires many data manipulations. Sometimes such data manipulation involves the decomposition of data elements. For example, an employee's date of birth can be subdivided into a day, a month, and a year. A product manufacturing code (e.g., SE-05-2-09-1234-1-3/12/18-19:26:48) can be designed to record the manufacturing region, plant, shift, production line, employee number, date, and time. For years, conventional programming languages have had special functions that enabled programmers to perform data transformations like the preceding data decompositions. If you know a modern programming language, it is very likely that the SQL functions in this section will look familiar.

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

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

Note

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

7-9a Date and Time Functions

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

Because date/time functions differ from vendor to vendor, this section will cover basic date/time functions for MS Access, SQL Server, and Oracle. Table 7.8 shows a list of selected MS Access and SQL Server date/time functions.

TABLE 7.8

SELECTED MS ACCESS AND SQL SERVER DATA/TIME FUNCTIONS

FUNCTION	EXAMPLE(S)
CONVERT (MS SQL Server) Convert can be used to perform a wide array of data type conversions as discussed next. It can also be used to format date data. Syntax: CONVERT(varchar(length), date_value, fmt_code) fmt_code = format used; can be: 1: MM/DD/YY 101: MM/DD/YYYY 2: YY.MM.DD 102: YYYY.MM.DD 3: DD/MM/YY 103: DD/MM/YYYY	Displays the product code and date the product was last received into stock for all products: SELECT P_CODE, CONVERT(VARCHAR(8), P_INDATE, 1) FROM PRODUCT; SELECT P_CODE, CONVERT(VARCHAR(10), P_INDATE, 102) FROM PRODUCT;
YEAR Returns a four-digit year Syntax: YEAR(date_value)	Lists all employees born in 1982: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, YEAR(EMP_DOB) AS YEAR FROM EMPLOYEE WHERE YEAR(EMP_DOB) = 1982;
MONTH Returns a two-digit month code Syntax: MONTH(date_value)	Lists all employees born in November: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, MONTH(EMP_DOB) AS MONTH FROM EMPLOYEE WHERE MONTH(EMP_DOB) = 11;
DAY Returns the number of the day Syntax: DAY(date_value)	Lists all employees born on the 14th day of the month: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, DAY(EMP_DOB) AS DAY FROM EMPLOYEE WHERE DAY(EMP_DOB) = 14;
DATE() MS Access **GETDATE() SQL Server** Returns today's date	Lists how many days are left until Christmas: SELECT #25-Dec-2018# – DATE(); Note two features: • There is no FROM clause, which is acceptable in Access and MS SQL Server. • The Christmas date is enclosed in number signs (#) because you are doing date arithmetic. In MS SQL Server: Use GETDATE() to get the current system date. To compute the difference between dates, use the DATEDIFF function (see below).
DATEADD SQL Server Adds a number of selected time periods to a date Syntax: DATEADD(datepart, number, date)	Adds a number of dateparts to a given date. Dateparts can be minutes, hours, days, weeks, months, quarters, or years. For example: SELECT DATEADD(day,90, P_INDATE) AS DueDate FROM PRODUCT; The preceding example adds 90 days to P_INDATE. In MS Access, use the following: SELECT P_INDATE+90 AS DueDate FROM PRODUCT;

TABLE 7.8 (CONTINUED)

SELECTED MS ACCESS AND SQL SERVER DATA/TIME FUNCTIONS

DATEDIFF SQL Server Subtracts two dates Syntax: DATEDIFF(datepart, startdate, enddate)	Returns the difference between two dates expressed in a selected datepart. For example: SELECT DATEDIFF(day, P_INDATE, GETDATE()) AS DaysAgo FROM PRODUCT; In MS Access, use the following: SELECT DATE() - P_INDATE AS DaysAgo FROM PRODUCT;

Table 7.9 shows the equivalent date/time functions used in Oracle. Note that Oracle uses the same function (TO_CHAR) to extract the various parts of a date. Also, another function (TO_DATE) is used to convert character strings to a valid Oracle date format that can be used in date arithmetic.

TABLE 7.9

SELECTED ORACLE DATE/TIME FUNCTIONS

FUNCTION	EXAMPLE(S)
TO_CHAR Returns a character string or a formatted string from a date value Syntax: TO_CHAR(date_value, fmt) fmt = format used; can be: MONTH: name of month MON: three-letter month name MM: two-digit month name D: number for day of week DD: number for day of month DAY: name of day of week YYYY: four-digit year value YY: two-digit year value	Lists all employees born in 1982: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, TO_CHAR(EMP_DOB, 'YYYY') AS YEAR FROM EMPLOYEE WHERE TO_CHAR(EMP_DOB, 'YYYY') = '1982'; Lists all employees born in November: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, TO_CHAR(EMP_DOB, 'MM') AS MONTH FROM EMPLOYEE WHERE TO_CHAR(EMP_DOB, 'MM') = '11'; Lists all employees born on the 14th day of the month: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, TO_CHAR(EMP_DOB, 'DD') AS DAY FROM EMPLOYEE WHERE TO_CHAR(EMP_DOB, 'DD') = '14';
TO_DATE Returns a date value using a character string and a date format mask; also used to translate a date between formats Syntax: TO_DATE(char_value, fmt) fmt = format used; can be: MONTH: name of month MON: three-letter month name MM: two-digit month name D: number for day of week DD: number for day of month DAY: name of day of week YYYY: four-digit year value YY: two-digit year value	Lists the approximate age of employees on the company's tenth anniversary date (11/25/2018): SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, '11/25/2018' AS ANIV_DATE, (TO_DATE('11/25/2018','MM/DD/YYYY') - EMP_DOB)/365 AS YEARS FROM EMPLOYEE ORDER BY YEARS; Note the following: • '11/25/2018' is a text string, not a date. • The TO_DATE function translates the text string to a valid Oracle date used in date arithmetic. How many days are there between Thanksgiving and Christmas 2018? SELECT TO_DATE('2018/12/25','YYYY/MM/DD') – TO_DATE('NOVEMBER 27, 2018','MONTH DD, YYYY') FROM DUAL; Note the following: • The TO_DATE function translates the text string to a valid Oracle date used in date arithmetic. • DUAL is Oracle's pseudo-table, used only for cases in which a table is not really needed.

TABLE 7.9 (CONTINUED)

SELECTED ORACLE DATE/TIME FUNCTIONS

SYSDATE Returns today's date	Lists how many days are left until Christmas: SELECT TO_DATE('25-Dec-2018','DD-MON-YYYY') - SYSDATE FROM DUAL; Notice two things: • DUAL is Oracle's pseudo-table, used only for cases in which a table is not really needed. • The Christmas date is enclosed in a TO_DATE function to translate the date to a valid date format.
ADD_MONTHS Adds a number of months or years to a date Syntax: ADD_MONTHS(date_value, n) n = number of months	Lists all products with their expiration date (two years from the purchase date): SELECT P_CODE, P_INDATE, ADD_MONTHS(P_INDATE,24) FROM PRODUCT ORDER BY ADD_MONTHS(P_INDATE,24);
LAST_DAY Returns the date of the last day of the month given in a date Syntax: LAST_DAY(date_value)	Lists all employees who were hired within the last seven days of a month: SELECT EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE FROM EMPLOYEE WHERE EMP_HIRE_DATE >=LAST_DAY(EMP_HIRE_DATE)-7;

Table 7.10 shows the equivalent functions for MySQL.

TABLE 7.10

SELECTED MYSQL DATE/TIME FUNCTIONS

FUNCTION	EXAMPLE(S)
Date_Format Returns a character string or a formatted string from a date value Syntax: DATE_FORMAT(date_value, fmt) fmt = format used; can be: %M: name of month %m: two-digit month number %b: abbreviated month name %d: number of day of month %W: weekday name %a: abbreviated weekday name %Y: four-digit year %y: two-digit year	Displays the product code and date the product was last received into stock for all products: SELECT P_CODE, DATE_FORMAT(P_INDATE, '%m/%d/%y') FROM PRODUCT; SELECT P_CODE, DATE_FORMAT(P_INDATE, '%M %d, %Y') FROM PRODUCT;
YEAR Returns a four-digit year Syntax: YEAR(date_value)	Lists all employees born in 1982: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, YEAR(EMP_DOB) AS YEAR FROM EMPLOYEE WHERE YEAR(EMP_DOB) = 1982;

TABLE 7.10 (CONTINUED)

SELECTED MYSQL DATE/TIME FUNCTIONS

MONTH Returns a two-digit month code Syntax: MONTH(date_value)	Lists all employees born in November: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, MONTH(EMP_DOB) AS MONTH FROM EMPLOYEE WHERE MONTH(EMP_DOB) = 11;
DAY Returns the number of the day Syntax: DAY(date_value)	Lists all employees born on the 14th day of the month: SELECT EMP_LNAME, EMP_FNAME, EMP_DOB, DAY(EMP_DOB) AS DAY FROM EMPLOYEE WHERE DAY(EMP_DOB) = 14;
ADDDATE Adds a number of days to a date Syntax: ADDDATE(date_value, n) n = number of days **DATE_ADD** Adds a number of days, months, or years to a date. This is similar to ADDDATE except it is more robust. It allows the user to specify the date unit to add. Syntax: DATE_ADD(date, INTERVAL n unit) n = number to add unit = date unit, can be: DAY: add n days WEEK: add n weeks MONTH: add n months YEAR: add n years	List all products with the date they will have been on the shelf for 30 days. SELECT P_CODE, P_INDATE, ADDDATE(P_INDATE, 30) FROM PRODUCT ORDER BY ADDDATE(P_INDATE, 30); Lists all products with their expiration date (two years from the purchase date): SELECT P_CODE, P_INDATE, DATE_ADD(P_INDATE, INTERVAL 2 YEAR) FROM PRODUCT ORDER BY DATE_ADD(P_INDATE, INTERVAL 2 YEAR);
LAST_DAY Returns the date of the last day of the month given in a date Syntax: LAST_DAY(date_value)	Lists all employees who were hired within the last seven days of a month: SELECT EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE FROM EMPLOYEE WHERE EMP_HIRE_DATE >= DATE_ADD(LAST_DAY (EMP_HIRE_DATE), INTERVAL -7 DAY);

7-9b Numeric Functions

Numeric functions can be grouped in many different ways, such as algebraic, trigonometric, and logarithmic. In this section, you will learn two very useful functions. Do not confuse the SQL aggregate functions you saw earlier in this chapter with the numeric functions in this section. The first group operates over a set of values (multiple rows—hence, the name *aggregate functions*), while the numeric functions covered here operate over a single row. Numeric functions take one numeric parameter and return one value. Table 7.11 shows a selected group of available numeric functions.

TABLE 7.11

SELECTED NUMERIC FUNCTIONS

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

7-9c String Functions

String manipulations are among the most-used functions in programming. If you have ever created a report using any programming language, you know the importance of properly concatenating strings of characters, printing names in uppercase, or knowing the length of a given attribute. Table 7.12 shows a subset of useful string manipulation functions.

TABLE 7.12

SELECTED STRING FUNCTIONS

FUNCTION	EXAMPLE(S)
Concatenation **|| Oracle** **+ Access and MS SQL Server** **& Access** **CONCAT() MySQL** Concatenates data from two different character columns and returns a single column. Syntax: strg_value || strg_value strg_value + strg_value strg_value & strg_value CONCAT(strg_value, strg_value) The CONCAT function can only accept two string values so nested CONCAT functions are required when more than two values are to be concatenated.	Lists all employee names (concatenated). In Oracle, use the following: SELECT EMP_LNAME || ', ' || EMP_FNAME AS NAME FROM EMPLOYEE; In Access and MS SQL Server, use the following: SELECT EMP_LNAME + ', ' + EMP_FNAME AS NAME FROM EMPLOYEE; In MySQL, use the following: SELECT CONCAT(CONCAT(EMP_LNAME, ', '), EMP_FNAME AS NAME FROM EMPLOYEE;

TABLE 7.12 (CONTINUED)

SELECTED STRING FUNCTIONS

FUNCTION	EXAMPLE(S)
UPPER Oracle, MS SQL Server, and MySQL **UCASE MySQL and Access** **LOWER Oracle, MS SQL Server, and MySQL** **LCASE MySQL and Access** Returns a string in all capital or all lowercase letters Syntax: UPPER(strg_value) UCASE(strg_value) LOWER(strg_value) LCASE(strg_value)	Lists all employee names in all capital letters (concatenated). In Oracle, use the following: SELECT UPPER(EMP_LNAME || ', ' || EMP_FNAME) AS NAME FROM EMPLOYEE; In MS SQL Server, use the following: SELECT UPPER(EMP_LNAME + ', ' + EMP_FNAME) AS NAME FROM EMPLOYEE; In Access, use the following: SELECT UCASE(EMP_LNAME & ', ' & EMP_FNAME) AS NAME FROM EMPLOYEE; In MySQL, use the following: SELECT UPPER(CONCAT(CONCAT(EMP_LNAME, ', '), EMP_FNAME AS NAME FROM EMPLOYEE; Lists all employee names in all lowercase letters (concatenated). In Oracle, use the following: SELECT LOWER(EMP_LNAME || ', ' || EMP_FNAME) AS NAME FROM EMPLOYEE; In MS SQL Server, use the following: SELECT LOWER(EMP_LNAME + ', ' + EMP_FNAME) AS NAME FROM EMPLOYEE; In Access, use the following: SELECT LCASE(EMP_LNAME & ', ' & EMP_FNAME) AS NAME FROM EMPLOYEE; In MySQL, use the following: SELECT LOWER(CONCAT(CONCAT(EMP_LNAME, ', '), EMP_FNAME AS NAME FROM EMPLOYEE;
SUBSTRING Returns a substring or part of a given string parameter Syntax: SUBSTR(strg_value, p, l) Oracle and MySQL SUBSTRING(strg_value,p,l) MS SQL Server and MySQL MID(strg_value,p,l) Access p = start position l = length of characters If the length of characters is omitted, the functions will return the remainder of the string value.	Lists the first three characters of all employee phone numbers. In Oracle or MySQL, use the following: SELECT EMP_PHONE, SUBSTR(EMP_PHONE,1,3) AS PREFIX FROM EMPLOYEE; In MS SQL Server or MySQL, use the following: SELECT EMP_PHONE, SUBSTRING(EMP_PHONE,1,3) AS PREFIX FROM EMPLOYEE; In Access, use the following: SELECT EMP_PHONE, MID(EMP_PHONE, 1,3) AS PREFIX FROM EMPLOYEE;
LENGTH Returns the number of characters in a string value Syntax: LENGTH(strg_value) Oracle and MySQL LEN(strg_value) MS SQL Server and Access	Lists all employee last names and the length of their names in descending order by last name length. In Oracle and MySQL, use the following: SELECT EMP_LNAME, LENGTH(EMP_LNAME) AS NAMESIZE FROM EMPLOYEE; In MS Access and SQL Server, use the following: SELECT EMP_LNAME, LEN(EMP_LNAME) AS NAMESIZE FROM EMPLOYEE;

7-9d Conversion Functions

Conversion functions allow you to take a value of a given data type and convert it to the equivalent value in another data type. In Section 7-9a, you learned about two basic Oracle SQL conversion functions: TO_CHAR and TO_DATE. Note that the TO_CHAR function takes a date value and returns a character string representing a day, a month, or a year. In the same way, the TO_DATE function takes a character string representing a date and returns an actual date in Oracle format. SQL Server uses the CAST and CONVERT functions to convert one data type to another. A summary of the selected functions is shown in Table 7.13.

TABLE 7.13

SELECTED CONVERSION FUNCTIONS

FUNCTION	EXAMPLE(S)
Numeric or Date to Character: **TO_CHAR Oracle** **CAST Oracle, MS SQL Server, MySQL** **CONVERT MS SQL Server, MySQL** **CSTR Access** Returns a character string from a numeric or date value. Syntax: TO_CHAR(value-to-convert, fmt) fmt = format used; can be: 9 = displays a digit 0 = displays a leading zero , = displays the comma . = displays the decimal point $= displays the dollar sign B = leading blank S = leading sign MI = trailing minus sign CAST (value-to-convert AS char(length)) Note that Oracle and MS SQL Server can use CAST to convert the numeric data into fixed length or variable length character data type. MySQL cannot CAST into variable length character data, only fixed length. MS SQL Server: CONVERT(varchar(length), value-to-convert) MySQL: CONVERT(value-to-convert, char(length)) The primary difference between CAST and CONVERT is that CONVERT can also be used to change the character set of the data. CSTR(value-to-convert)	Lists all product prices, product received date, and percent discount using formatted values. TO_CHAR: SELECT P_CODE, TO_CHAR(P_PRICE,'999.99') AS PRICE, TO_CHAR(P_INDATE, 'MM/DD/YYYY') AS INDATE, TO_CHAR(P_DISCOUNT,'0.99') AS DISC FROM PRODUCT; CAST in Oracle and MS SQL Server: SELECT P_CODE, CAST(P_PRICE AS VARCHAR(8)) AS PRICE, CAST(P_INDATE AS VARCHAR(20)) AS INDATE, CAST(P_DISCOUNT AS VARCHAR(4)) AS DISC FROM PRODUCT; CAST in MySQL: SELECT P_CODE, CAST(P_PRICE AS CHAR(8)) AS PRICE, CAST(P_INDATE AS CHAR(20)) AS INDATE, CAST(P_DISCOUNT AS CHAR(4)) AS DISC FROM PRODUCT; CONVERT in MS SQL Server: SELECT P_CODE, CONVERT(VARCHAR(8), P_PRICE) AS PRICE, CONVERT(VARCHAR(20), P_INDATE) AS INDATE, CONVERT(VARCHAR(4), P_DISC) AS DISC FROM PRODUCT; CONVERT in MySQL: SELECT P_CODE, CONVERT(P_PRICE, CHAR(8)) AS PRICE, CONVERT(P_INDATE, CHAR(20)) AS INDATE, CONVERT(P_DISC, CHAR(4)) AS DISC FROM PRODUCT; CSTR in Access: SELECT P_CODE, CSTR(P_PRICE) AS PRICE, CSTR(P_INDATE) AS INDATE, CSTR(P_DISC) AS DISCOUNT FROM PRODUCT;

TABLE 7.13 (CONTINUED)

SELECTED CONVERSION FUNCTIONS

FUNCTION	EXAMPLE(S)
String to Number: **TO_NUMBER Oracle** **CAST Oracle, MS SQL Server, MySQL** **CONVERT MS SQL Server, MySQL** **CINT Access** **CDEC Access** Returns a number from a character string Syntax: Oracle: TO_NUMBER(char_value, fmt) fmt = format used; can be: 9 = indicates a digit B = leading blank S = leading sign MI = trailing minus sign CAST (value-to-convert as numeric-data type) Note that in addition to the INTEGER and DECIMAL(l,d) data types, Oracle supports NUMBER and MS SQL Server supports NUMERIC. MS SQL Server: CONVERT(value-to-convert, decimal(l,d)) MySQL: CONVERT(value-to-convert, decimal(l,d)) Other than the data type to be converted into, these functions operate the same as described above. CINT in Access returns the number in the integer data type, while CDEC returns decimal data type.	Converts text strings to numeric values when importing data to a table from another source in text format; for example, the query shown here uses the TO_NUMBER function to convert text formatted to Oracle default numeric values using the format masks given. TO_NUMBER: SELECT TO_NUMBER('–123.99', '$999.99'), TO_NUMBER('99.78–','B999.99MI') FROM DUAL; CAST: SELECT CAST('–123.99' AS DECIMAL(8,2)), CAST('–99.78' AS DECIMAL(8,2)); The CAST function does not support the trailing sign on the character string. CINT and CDEC: SELECT CINT('–123'), CDEC('–123.99');
CASE Oracle, MS SQL Server, MySQL **DECODE Oracle** **SWITCH Access** Compares an attribute or expression with a series of values and returns an associated value or a default value if no match is found Syntax: DECODE: DECODE(e, x, y, d) e = attribute or expression x = value with which to compare e y = value to return in e = x d = default value to return if e is not equal to x CASE: CASE When condition THEN value1 ELSE value2 END **SWITCH:** **SWITCH(e1, x, e2, y, TRUE, d)** e1 = comparison expression x = value to return if e1 is true e2 = comparison expression y = value to return if e2 is true TRUE = keyword indicating the next value is the default d = default value to return if none of the expressions were true	The following example returns the sales tax rate for specified states: Compares V_STATE to 'CA'; if the values match, it returns .08. Compares V_STATE to 'FL'; if the values match, it returns .05. Compares V_STATE to 'TN'; if the values match, it returns .085. If there is no match, it returns 0.00 (the default value). SELECT V_CODE, V_STATE, DECODE(V_STATE,'CA',.08,'FL',.05, 'TN',.085, 0.00) AS TAX FROM VENDOR; CASE: SELECT V_CODE, V_STATE, CASE WHEN V_STATE = 'CA' THEN .08 WHEN V_STATE = 'FL' THEN .05 WHEN V_STATE = 'TN' THEN .085 ELSE 0.00 END AS TAX FROM VENDOR; SWITCH: SELECT V_CODE, V_STATE, SWITCH(V_STATE ='CA',.08, V_STATE = 'FL',.05, V_STATE = 'TN',.085, TRUE, 0.00) AS TAX FROM VENDOR;

7-10 Relational Set Operators

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

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

Note

The SQL standard defines the operations that all DBMSs must perform on data, but it leaves the implementation details to the DBMS vendors. Therefore, some advanced SQL features might not work on all DBMS implementations. Also, some DBMS vendors might implement additional features not found in the SQL standard. The SQL standard defines UNION, INTERSECT, and EXCEPT as the keywords for the UNION, INTERSECT, and DIFFERENCE relational operators, and these are the names used in MS SQL Server. However, Oracle uses MINUS as the name of the DIFFERENCE operator instead of EXCEPT. Other RDBMS vendors might use a different operator name or might not implement a given operator at all. For example, Access and MySQL do not have direct support for INTERSECT or DIFFERENCE operations because that functionality can be achieved using combinations of joins and subqueries. To learn more about the ANSI/ISO SQL standards and find out how to obtain the latest standard documents in electronic form, check the ANSI website (*www.ansi.org*).

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

Note

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

7-10a UNION

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

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

query UNION *query*

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

To demonstrate the use of the UNION statement in SQL, use the CUSTOMER and CUSTOMER_2 tables in the Ch07_SaleCo database. To show the combined CUSTOMER and CUSTOMER_2 records without duplicates, the UNION query is written as follows:

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

Figure 7.59 shows the contents of the CUSTOMER and CUSTOMER_2 tables and the result of the UNION query. Although MS Access is used to show the results here, similar results can be obtained with Oracle, MS SQL Server, and MySQL.

FIGURE 7.59 UNION QUERY RESULTS

Database name: Ch07_SaleCo

Table name: CUSTOMER

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

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

CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
Brown	James	G	615	297-1228
Dunne	Leona	K	713	894-1238
Farriss	Anne	G	713	382-7185
Hernandez	Carlos	J	723	123-7654
Lewis	Marie	J	734	332-1789
McDowell	George		723	123-7768
O'Brian	Amy	B	713	442-3381
Olowski	Paul	F	615	894-2180
Orlando	Myron		615	222-1672
Ramas	Alfred	A	615	844-2573
Smith	Kathy	W	615	894-2285
Smith	Olette	K	615	297-3809
Terrell	Justine	H	615	322-9870
Tyron	Khaleed	G	723	123-6876
Williams	George		615	290-2556

Table name: CUSTOMER_2

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
345	Terrell	Justine	H	615	322-9870
347	Olowski	Paul	F	615	894-2180
351	Hernandez	Carlos	J	723	123-7654
352	McDowell	George		723	123-7768
355	Tryon	Khaleed	G	723	123-6876
356	Lewis	Marie	J	734	332-1789
359	Dunne	Leona	K	713	894-1238

Note the following in Figure 7.59:

- The CUSTOMER table contains 10 rows, while the CUSTOMER_2 table contains seven rows.
- Customers Dunne and Olowski are included in the CUSTOMER table as well as the CUSTOMER_2 table.

- The UNION query yields 15 records because the duplicate records of customers Dunne and Olowski are not included. In short, the UNION query yields a unique set of records.

Note

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

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

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

7-10b UNION ALL

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

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

Running the preceding UNION ALL query produces the result shown in Figure 7.60.

FIGURE 7.60 UNION ALL QUERY RESULTS

Database name: Ch07_SaleCo

Table name: CUSTOMER

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

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

CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
Ramas	Alfred	A	615	844-2573
Dunne	Leona	K	713	894-1238
Smith	Kathy	W	615	894-2285
Olowski	Paul	F	615	894-2180
Orlando	Myron		615	222-1672
O'Brian	Amy	B	713	442-3381
Brown	James	G	615	297-1228
Williams	George		615	290-2556
Farriss	Anne	G	713	382-7185
Smith	Olette	K	615	297-3809
Terrell	Justine	H	615	322-9870
Tryon	Khaleed	G	723	123-6876
Olowski	Paul	F	615	894-2180
Hernandez	Carlos	J	723	123-7654
McDowell	George		723	123-7768
Tryon	Khaleed	G	723	123-6876
Lewis	Marie	J	734	332-1789
Dunne	Leona	K	713	894-1238

Table name: CUSTOMER_2

CUS_CODE	CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
345	Terrell	Justine	H	615	322-9870
347	Olowski	Paul	F	615	894-2180
351	Hernandez	Carlos	J	723	123-7654
352	McDowell	George		723	123-7768
355	Tryon	Khaleed	G	723	123-6876
356	Lewis	Marie	J	734	332-1789
359	Dunne	Leona	K	713	894-1238

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

7-10c INTERSECT

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

query INTERSECT *query*

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

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

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

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

Figure 7.61 shows the result.

FIGURE 7.61 INTERSECT QUERY RESULTS

CUS_CODE
10012
10014

7-10d EXCEPT (MINUS)

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

query EXCEPT *query*

and

query MINUS *query*

For example, if the SaleCo managers want to know which customers in the CUSTOMER table are not found in the CUSTOMER_2 table, they can use the following command in Oracle (see Figure 7.62).

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

FIGURE 7.62 CUSTOMER MINUS CUSTOMER_2 QUERY RESULTS

CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
Ramas	Alfred	A	615	844-2573
Smith	Kathy	W	615	894-2285
Orlando	Myron		615	222-1672
O'Brian	Amy	B	713	442-3381
Brown	James	G	615	297-1228
Williams	George		615	290-2556
Farriss	Anne	G	713	382-7185
Smith	Olette	K	615	297-3809

If the managers want to know which customers in the CUSTOMER_2 table are not found in the CUSTOMER table, they merely switch the table designations (see Figure 7.63):

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

FIGURE 7.63 CUSTOMER_2 MINUS CUSTOMER QUERY RESULTS

CUS_LNAME	CUS_FNAME	CUS_INITIAL	CUS_AREACODE	CUS_PHONE
Terrell	Justine	H	615	322-9870
Hernandez	Carlos	J	723	123-7654
McDowell	George		723	123-7768
Tirpin	Khaleed	G	723	123-9876
Lewis	Marie	J	734	332-1789

Users of MS SQL Server would substitute the keyword EXCEPT in place of MINUS, but otherwise the syntax is exactly the same. You can extract useful information by combining MINUS with various clauses such as WHERE. For example, the following query returns the customer codes for all customers in area code 615 minus the ones who have made purchases, leaving the customers in area code 615 who have not made purchases.

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

7-10e Syntax Alternatives

If your DBMS does not support the INTERSECT or EXCEPT (MINUS) statements, you can use alternative syntax to achieve the same output. For example, the INTERSECT query:

```
SELECT      CUS_AREACODE FROM CUSTOMER
INTERSECT
SELECT      V_AREACODE FROM VENDOR;
```

can be reproduced without using the INTERSECT operator by the following:

```
SELECT      DISTINCT CUS_AREACODE
FROM        CUSTOMER JOIN VENDOR ON CUS_AREACODE = V_AREACODE;
```

SQL allows programmers to solve a given problem in a variety of ways. IN and NOT IN subqueries can be used to obtain results for other INTERSECT and MINUS queries. For example, the following query will produce the same results as the INTERSECT query shown in Figure 7.61:

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

Using the same alternative to the MINUS statement, you can generate the output for the EXCEPT query shown above by entering the following:

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


7-11 Crafting SELECT Queries

As you have seen in this chapter, the SQL language is both simple and complex. Each clause and function on its own is simple and performs a well-defined task. However, because of the flexibility of the SQL language, combining the appropriate clauses and functions to satisfy an information request can become rather complex. When attempting to craft a query, the following are useful suggestions to keep in mind.

7-11a Know Your Data

The importance of understanding the data model that you are working in cannot be overstated. Databases in academic courses are normally well designed, well structured, and follow best practices. Real-world databases are messy. Table and attribute names are often cryptic, confusing, and nonstandardized. Tables may not have appropriate constraints enforced, and, in some cases, may not even have a defined primary key! Finding tables of related data that do not have a foreign key to implement that relationship is not uncommon.

The problem is not that practicing database professionals are poor at doing their jobs. Remember, most database systems remain in service in an organization for decades. As the business changes, grows, contracts, merges, and splits over many years, the internal systems must be adapted and changed. These changes often involve compromises that become institutionalized within the system. For example, the authors are familiar with a database in a healthcare company that, due to a merger of companies many years ago, has multiple tables that contain data related to patient treatment. In one table, the attribute PID (the patient ID) is an identifier for the person receiving treatment. In the second table, the attribute PID is an identifier for the person against whose insurance the treatment is being billed. SQL programmers in that environment deal with a large number of confusing table and column names.

It can be difficult to grasp a new environment, but a SQL programmer who does not know the data model he or she is working in will not know what data is available to answer questions, how the data are related, or how to access it. As a new database professional, you may find yourself thrown into an environment where you are responsible for working with hundreds of tables. It will take time, but be diligent in working toward learning and understanding the data.

7-11b Know the Problem

Just as important as it is to understand the data model, it is equally important to understand the question you are attempting to answer. Information reporting requests will come from a range of sources. Some requests are one-time events, and some will become a part of on-going operations within an application or data analysis process. Information requests are often ambiguous and subject to multiple interpretations, even if the person making the request does not realize it. For example, consider a scenario in which a marketing manager wants to know the average price for which we have sold a particular product. Assume there have been ten sales of the product with the following values: $10, $10, $10, $20, $10, $10, $25, $10, $10, and $10. Which did the marketing manager want?

- The average price for all of the sales that have occurred:

  10 + 10 + 10 + 10 + 10 + 10 + 10 + 10 + 20 + 30 = 130 / 10 = $13

  Coded as: SELECT AVG(SALE_PRICE)

- The average of the prices at which any sale has occurred:

  10 + 20 + 30 = 60 / 3 = $20

  Coded as: SELECT AVG(DISTINCT SALE_PRICE)

Even with only ten rows of data, a clear difference in the possible answers quickly becomes apparent. This issue is crucial because the marketing manager may not have considered how ambiguous the request was. When presented with an answer, decisions will be made based on the information presented. If there was miscommunication between the manager and the programmer, the business may make a poor decision with significant consequences.

7-11c Build One Clause at a Time

Once you understand the problem and you know your data model so that you can map the problem to the data, you can build the actual query. Remembering how the clauses in a SELECT query work together, it may be helpful to build your clauses in the following order:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT
- ORDER BY

Map the requirements to the data model to understand which tables contain the data that will be required. For performance reasons, use the smallest set of related tables possible to answer your query. For example, if a query requires only the vendor code and product description attributes, a look at Figure 7.1 reveals that the PRODUCT table contains both of those attributes. Therefore, there would be no reason to include the VENDOR table in the query. Write the appropriate FROM clause to join the required tables. You can start with a simple SELECT * for the SELECT column list so that you can test that your FROM clause is retrieving the data that you had intended. Ensure that you are using the correct outer joins when those are needed. Also for performance reasons, do not use an outer join when an inner join will suffice.

Next, decide if all of the rows returned by the FROM clause are desired in your result. If not, write one or more criteria in the WHERE clause that can be used to restrict the data to only the rows that meet the requirements. If all of the rows are required, then a WHERE clause is not needed.

Will your query need to return an aggregate value? If so, determine the appropriate attributes on which to group the data. If no aggregate will be returned, then the GROUP BY clause is not needed. If the GROUP BY clause is not needed, then the HAVING clause is not needed either. Remember, the HAVING clause is used to restrict groups. If there are no groups, then HAVING is not needed. If the query does use a GROUP BY clause, then decide if all of the groups should be returned in the answer. If so, then a HAVING clause is not needed. If some groups should not be included in the result, then write criteria in the HAVING clause that can be used to restrict the groups to only the groups that are of interest. Also recall that since the HAVING clause cannot restrict individual rows in a group, it must apply to the whole group; the criteria should include an aggregate function. If you can write a criteria that applies to the whole group but

does not contain an aggregate function, then that criteria should probably have been included in the WHERE clause instead.

Next, specify the attributes and aggregates that should be returned in the SELECT column list. If any derived attributes need to be returned, then remember to include the formulas to calculate them in the SELECT. Also, consider if the DISTINCT keyword is needed. For performance reasons, do not include DISTINCT if it is not needed. If the query is returning duplicate rows of output that should be suppressed, then place DISTINCT immediately after the SELECT keyword. Note that this should not normally be the case if an aggregate function is being returned since the GROUP BY clause will combine any duplicates into a single collection that is reduced to one row by the aggregate function. However, if an aggregate function is being used, consider whether or not duplicate values should be suppressed during the calculation of the aggregate, and if so, then include DISTINCT inside the aggregate function.

Finally, consider the sorting of the rows in the final output. For performance reasons, if the order of the rows in the final output does not matter, then omit the ORDER BY clause. However, if the ordering matters, then determine the attribute or attributes that should be used for sorting. If, during the construction of any clause in the SELECT query, you determine that the data must be preprocessed before the query can use it appropriately, then a subquery may be needed.

Summary

- SQL commands can be divided into two overall categories: data definition language (DDL) commands and data manipulation language (DML) commands.
- The ANSI standard data types are supported by all RDBMS vendors in different ways. The basic data types are NUMBER, NUMERIC, INTEGER, CHAR, VARCHAR, and DATE.
- The SELECT statement is the main data retrieval command in SQL. A SELECT statement has the following syntax:

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

- The column list represents one or more column names separated by commas. The column list may also include computed columns, aliases, and aggregate functions. A computed column is represented by an expression or formula (e.g., P_PRICE * P_QOH). The FROM clause contains a list of table names.
- Operations that join tables can be classified as inner joins and outer joins. An inner join is the traditional join in which only rows that meet a given criterion are selected. An outer join returns the matching rows as well as the rows with unmatched attribute values for one table or both tables to be joined.
- A natural join returns all rows with matching values in the matching columns and eliminates duplicate columns. This style of query is used when the tables share a common attribute with a common name. One important difference between the syntax for a natural join and for the old-style join is that the natural join does not require

the use of a table qualifier for the common attributes. In practice, natural joins are often discouraged because the common attribute is not specified within the command, making queries more difficult to understand and maintain.

- Joins may use keywords such as USING and ON. If the USING clause is used, the query will return only the rows with matching values in the column indicated in the USING clause; that column must exist in both tables. If the ON clause is used, the query will return only the rows that meet the specified join condition.
- The ORDER BY clause is used to sort the output of a SELECT statement. The ORDER BY clause can sort by one or more columns and can use either ascending or descending order.
- The WHERE clause can be used with the SELECT, UPDATE, and DELETE statements to restrict the rows affected by the DDL command. The condition list represents one or more conditional expressions separated by logical operators (AND, OR, and NOT). The conditional expression can contain any comparison operators (=, >, <, >=, <=, and <>) as well as special operators (BETWEEN, IS NULL, LIKE, IN, and EXISTS).
- Aggregate functions (COUNT, MIN, MAX, and AVG) are special functions that perform arithmetic computations over a set of rows. The aggregate functions are usually used in conjunction with the GROUP BY clause to group the output of aggregate computations by one or more attributes. The HAVING clause is used to restrict the output of the GROUP BY clause by selecting only the aggregate rows that match a given condition.
- Subqueries and correlated queries are used when it is necessary to process data based on *other* processed data. That is, the query uses results that were previously unknown and that are generated by another query. Subqueries may be used with the FROM, WHERE, IN, and HAVING clauses in a SELECT statement. A subquery may return a single row or multiple rows.
- Most subqueries are executed in a serial fashion. That is, the outer query initiates the data request, and then the inner subquery is executed. In contrast, a correlated subquery is a subquery that is executed once for each row in the outer query. That process is similar to the typical nested loop in a programming language. A correlated subquery is so named because the inner query is related to the outer query—the inner query references a column of the outer subquery.
- SQL functions are used to extract or transform data. The most frequently used functions are date and time functions. The results of the function output can be used to store values in a database table, to serve as the basis for the computation of derived variables, or to serve as a basis for data comparisons. Function formats can be vendor-specific. Aside from time and date functions, there are numeric and string functions as well as conversion functions that convert one data format to another.
- SQL provides relational set operators to combine the output of two queries to generate a new relation. The UNION and UNION ALL set operators combine the output of two or more queries and produce a new relation with all unique (UNION) or duplicate (UNION ALL) rows from both queries. The INTERSECT relational set operator selects only the common rows. The EXCEPT (MINUS) set operator selects only the rows that are different. UNION, INTERSECT, and EXCEPT require union-compatible relations.
- Crafting effective and efficient SQL queries requires a great deal of skill. In order to successfully craft complex queries, the SQL programmer must understand the data with which he or she is working, and understand the problem to be solved. When struggling with the formulation of the query itself, building the query components in the order FROM, WHERE, GROUP BY, HAVING, SELECT, and ORDER BY can be helpful.

Key Terms

alias	FROM	recursive query
AND	GROUP BY	rules of precedence
AVG	HAVING	SELECT
BETWEEN	IN	set-oriented
Boolean algebra	IS NULL	subquery
cascading order sequence	LIKE	SUM
correlated subquery	MAX	transaction
COUNT	MIN	WHERE
cross join	NOT	wildcard character
DISTINCT	OR	
EXISTS	ORDER BY	

Review Questions

1. Explain why it would be preferable to use a DATE data type to store date data instead of a character data type.
2. Explain why the following command would create an error and what changes could be made to fix the error:

   SELECT V_CODE, SUM(P_QOH) FROM PRODUCT;
3. What is a cross join? Give an example of its syntax.
4. What three join types are included in the outer join classification?
5. Using tables named T1 and T2, write a query example for each of the three join types you described in Question 4. Assume that T1 and T2 share a common column named C1.
6. What is a recursive join?
7. Rewrite the following WHERE clause without the use of the IN special operator:

   WHERE V_STATE IN ('TN', 'FL', 'GA')
8. Explain the difference between an ORDER BY clause and a GROUP BY clause.
9. Explain why the following two commands produce different results:

   SELECT DISTINCT COUNT (V_CODE) FROM PRODUCT;

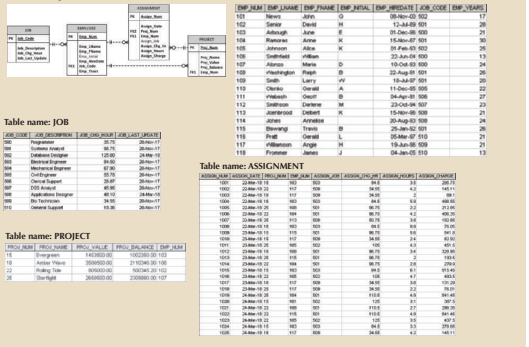
   SELECT COUNT (DISTINCT V_CODE) FROM PRODUCT;
10. What is the difference between the COUNT aggregate function and the SUM aggregate function?
11. In a SELECT query, what is the difference between a WHERE clause and a HAVING clause?
12. What is a subquery, and what are its basic characteristics?
13. What are the three types of results that a subquery can return?

14. What is a correlated subquery? Give an example.
15. Explain the difference between a regular subquery and a correlated subquery.
16. What does it mean to say that SQL operators are set-oriented?
17. The relational set operators UNION, INTERSECT, and EXCEPT (MINUS) work properly only when the relations are union-compatible. What does union-compatible mean, and how would you check for this condition?
18. What is the difference between UNION and UNION ALL? Write the syntax for each.
19. Suppose you have two tables: EMPLOYEE and EMPLOYEE_1. The EMPLOYEE table contains the records for three employees: Alice Cordoza, John Cretchakov, and Anne McDonald. The EMPLOYEE_1 table contains the records for employees John Cretchakov and Mary Chen. Given that information, list the query output for the UNION query.
20. Given the employee information in Question 19, list the query output for the UNION ALL query.
21. Given the employee information in Question 19, list the query output for the INTERSECT query.
22. Given the employee information in Question 19, list the query output for the EXCEPT (MINUS) query of EMPLOYEE to EMPLOYEE_1.
23. Suppose a PRODUCT table contains two attributes, PROD_CODE and VEND_CODE. Those two attributes have values of ABC, 125, DEF, 124, GHI, 124, and JKL, 123, respectively. The VENDOR table contains a single attribute, VEND_CODE, with values 123, 124, 125, and 126, respectively. (The VEND_CODE attribute in the PRODUCT table is a foreign key to the VEND_CODE in the VENDOR table.) Given that information, what would be the query output for:
    a. A UNION query based on the two tables?
    b. A UNION ALL query based on the two tables?
    c. An INTERSECT query based on the two tables?
    d. An EXCEPT (MINUS) query based on the two tables?
24. Why does the order of the operands (tables) matter in an EXCEPT (MINUS) query but not in a UNION query?
25. What MS Access and SQL Server function should you use to calculate the number of days between your birth date and the current date?
26. What Oracle function should you use to calculate the number of days between your birth date and the current date?
27. What string function should you use to list the first three characters of a company's EMP_LNAME values? Give an example using a table named EMPLOYEE. Provide examples for Oracle and SQL Server.
28. What two things must a SQL programmer understand before beginning to craft a SELECT query?

Problems

The Ch07_ConstructCo database stores data for a consulting company that tracks all charges to projects. The charges are based on the hours each employee works on each project. The structure and contents of the Ch07_ConstructCo database are shown in Figure P7.1.

FIGURE P7.1 THE CH07_CONSTRUCTCO DATABASE



Note that the ASSIGNMENT table in Figure P7.1 stores the JOB_CHG_HOUR values as an attribute (ASSIGN_CHG_HR) to maintain historical accuracy of the data. The JOB_CHG_HOUR values are likely to change over time. In fact, a JOB_CHG_HOUR change will be reflected in the ASSIGNMENT table. Naturally, the employee primary job assignment might also change, so the ASSIGN_JOB is also stored. Because those attributes are required to maintain the historical accuracy of the data, they are *not* redundant.

Given the structure and contents of the Ch07_ConstructCo database shown in Figure P7.1, use SQL commands to answer the following problems.

1. Write the SQL code required to list the employee number, last name, first name, and middle initial of all employees whose last names start with *Smith*. In other words, the rows for both Smith and Smithfield should be included in the listing. Sort the results by employee number. Assume case sensitivity.

2. Using the EMPLOYEE, JOB, and PROJECT tables in the Ch07_ConstructCo database, write the SQL code that will join the EMPLOYEE and PROJECT tables using EMP_NUM as the common attribute. Display the attributes shown in the results presented in Figure P7.2, sorted by project value.

FIGURE P7.2 THE QUERY RESULTS FOR PROBLEM 2

PROJ_NAME	PROJ_VALUE	PROJ_BALANCE	EMP_LNAME	EMP_FNAME	EMP_INITIAL	JOB_CODE	JOB_DESCRIPTION	JOB_CHG_HOUR
Rolling Tide	805000.00	500345.20	Senior	David	H	501	Systems Analyst	96.75
Evergreen	1453500.00	1002350.00	Arbough	June	E	500	Programmer	35.75
Starlight	2650500.00	2306880.00	Alonzo	Maria	D	500	Programmer	35.75
Amber Wave	3500500.00	2110346.00	Washington	Ralph	B	501	Systems Analyst	96.75

3. Write the SQL code that will produce the same information that was shown in Problem 2, but sorted by the employee's last name.
4. Write the SQL code that will list only the distinct project numbers in the ASSIGNMENT table, sorted by project number.
5. Write the SQL code to validate the ASSIGN_CHARGE values in the ASSIGNMENT table. Your query should retrieve the assignment number, employee number, project number, the stored assignment charge (ASSIGN_CHARGE), and the calculated assignment charge (calculated by multiplying ASSIGN_CHG_HR by ASSIGN_HOURS). Sort the results by the assignment number.
6. Using the data in the ASSIGNMENT table, write the SQL code that will yield the total number of hours worked for each employee and the total charges stemming from those hours worked, sorted by employee number. The results of running that query are shown in Figure P7.6.

FIGURE P7.6 TOTAL HOURS AND CHARGES BY EMPLOYEE

EMP_NUM	EMP_LNAME	SumOfASSIGN_HOURS	SumOfASSIGN_CHARGE
101	News	3.1	387.50
103	Arbough	19.7	1664.65
104	Ramoras	11.9	1218.70
105	Johnson	12.5	1382.50
108	Washington	8.3	840.15
113	Joenbrood	3.8	192.85
115	Bawangi	12.5	1276.75
117	Williamson	18.8	649.54

7. Write a query to produce the total number of hours and charges for each of the projects represented in the ASSIGNMENT table, sorted by project number. The output is shown in Figure P7.7.

FIGURE P7.7 TOTAL HOURS AND CHARGES BY PROJECT

PROJ_NUM	SumOfASSIGN_HOURS	SumOfASSIGN_CHARGE
15	20.5	1806.52
18	23.7	1544.80
22	27.8	2553.16
25	19.4	1668.15

8. Write the SQL code to generate the total hours worked and the total charges made by all employees. The results are shown in Figure P7.8.

FIGURE P7.8 TOTAL HOURS AND CHARGES, ALL EMPLOYEES

SumOfSumOfASSIGN_HOURS	SumOfSumOfASSIGN_CHARGE
90.6	7612.64

The structure and contents of the Ch07_SaleCo database are shown in Figure P7.9. Use this database to answer the following problems.

FIGURE P7.9 THE CH07_SALECO DATABASE

Relational diagram

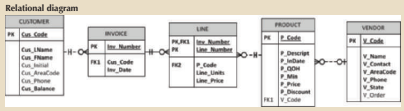


Table name: CUSTOMER

Table name: VENDOR

Table name: INVOICE

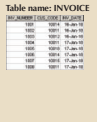


Table name: LINE

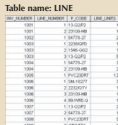


Table name: PRODUCT

9. Write a query to count the number of invoices.
10. Write a query to count the number of customers with a balance of more than $500.
11. Generate a listing of all purchases made by the customers, using the output shown in Figure P7.11 as your guide. Sort the results by customer code, invoice number, and product description.

FIGURE P7.11 LIST OF CUSTOMER PURCHASES

CUS_CODE	INV_NUMBER	INV_DATE	P_DESCRIPT	LINE_UNITS	LINE_PRICE
10011	1002	16-Jan-18	Rat-tail file, 1/8-in. fine	2	4.99
10011	1004	17-Jan-18	Claw hammer	2	9.95
10011	1004	17-Jan-18	Rat-tail file, 1/8-in. fine	3	4.99
10011	1008	17-Jan-18	Claw hammer	1	9.95
10011	1008	17-Jan-18	PVC pipe, 3.5-in., 8-ft	5	5.87
10011	1008	17-Jan-18	Steel matting, 4'x8'x1/6", .5" mesh	3	119.95
10012	1003	16-Jan-18	7.25-in. pwr. saw blade	5	14.99
10012	1003	16-Jan-18	B&D cordless drill, 1/2-in.	1	38.95
10012	1003	16-Jan-18	Hrd. cloth, 1/4-in., 2x50	1	39.95
10014	1001	16-Jan-18	7.25-in. pwr. saw blade	1	14.99
10014	1001	16-Jan-18	Claw hammer	1	9.95
10014	1006	17-Jan-18	1.25-in. metal screw, 25	3	6.99
10014	1006	17-Jan-18	B&D jigsaw, 12-in. blade	1	109.92
10014	1006	17-Jan-18	Claw hammer	1	9.95
10014	1006	17-Jan-18	Hicut chain saw, 16 in.	1	256.99
10015	1007	17-Jan-18	7.25-in. pwr. saw blade	2	14.99
10015	1007	17-Jan-18	Rat-tail file, 1/8-in. fine	1	4.99
10018	1005	17-Jan-18	PVC pipe, 3.5-in., 8-ft	12	5.87

12. Using the output shown in Figure P7.12 as your guide, generate a list of customer purchases, including the subtotals for each of the invoice line numbers. The subtotal is a derived attribute calculated by multiplying LINE_UNITS by LINE_PRICE. Sort the output by customer code, invoice number, and product description. Be certain to use the column aliases as shown in the figure.

FIGURE P7.12 SUMMARY OF CUSTOMER PURCHASES WITH SUBTOTALS

CUS_CODE	INV_NUMBER	P_DESCRIPT	Units Bought	Unit Price	Subtotal
10011	1002	Rat-tail file, 1/8-in. fine	2	4.99	9.98
10011	1004	Claw hammer	2	9.95	19.90
10011	1004	Rat-tail file, 1/8-in. fine	3	4.99	14.97
10011	1008	Claw hammer	1	9.95	9.95
10011	1008	PVC pipe, 3.5-in., 8-ft	5	5.87	29.35
10011	1008	Steel matting, 4'x8'x1/6", .5" mesh	3	119.95	359.85
10012	1003	7.25-in. pwr. saw blade	5	14.99	74.95
10012	1003	B&D cordless drill, 1/2-in.	1	38.95	38.95
10012	1003	Hrd. cloth, 1/4-in., 2x50	1	39.95	39.95
10014	1001	7.25-in. pwr. saw blade	1	14.99	14.99
10014	1001	Claw hammer	1	9.95	9.95
10014	1006	1.25-in. metal screw, 25	3	6.99	20.97
10014	1006	B&D jigsaw, 12-in. blade	1	109.92	109.92
10014	1006	Claw hammer	1	9.95	9.95
10014	1006	Hicut chain saw, 16 in.	1	256.99	256.99
10015	1007	7.25-in. pwr. saw blade	2	14.99	29.98
10015	1007	Rat-tail file, 1/8-in. fine	1	4.99	4.99
10018	1005	PVC pipe, 3.5-in., 8-ft	12	5.87	70.44

13. Write a query to display the customer code, balance, and total purchases for each customer. Total purchase is calculated by summing the line subtotals (as calculated in Problem 12) for each customer. Sort the results by customer code, and use aliases as shown in Figure P7.13.

FIGURE P7.13 CUSTOMER PURCHASE SUMMARY

CUS_CODE	CUS_BALANCE	Total Purchases
10011	0.00	444.00
10012	345.86	153.85
10014	0.00	422.77
10015	0.00	34.97
10018	216.55	70.44

14. Modify the query in Problem 13 to include the number of individual product purchases made by each customer. (In other words, if the customer's invoice is based on three products, one per LINE_NUMBER, you count three product purchases. Note that in the original invoice data, customer 10011 generated three invoices, which contained a total of six lines, each representing a product purchase.) Your output values must match those shown in Figure P7.14, sorted by customer code.

FIGURE P7.14 CUSTOMER TOTAL PURCHASE AMOUNTS AND NUMBER OF PURCHASES

CUS_CODE	CUS_BALANCE	Total Purchases	Number of Purchases
10011	0.00	444.00	6
10012	345.86	153.85	3
10014	0.00	422.77	6
10015	0.00	34.97	2
10018	216.55	70.44	1

15. Use a query to compute the total of all purchases, the number of purchases, and the average purchase amount made by each customer. Your output values must match those shown in Figure P7.15. Sort the results by customer code.

FIGURE P7.15 AVERAGE PURCHASE AMOUNT BY CUSTOMER

CUS_CODE	CUS_BALANCE	Total Purchases	Number of Purchases	Average Purchase Amount
10011	0.00	444.00	6	74.00
10012	345.86	153.85	3	51.28
10014	0.00	422.77	6	70.46
10015	0.00	34.97	2	17.48
10018	216.55	70.44	1	70.44

16. Create a query to produce the total purchase per invoice, generating the results shown in Figure P7.16, sorted by invoice number. The invoice total is the sum of the product purchases in the LINE that corresponds to the INVOICE.

FIGURE P7.16 INVOICE TOTALS

INV_NUMBER	Invoice Total
1001	24.94
1002	9.98
1003	153.85
1004	34.97
1005	70.44
1006	397.83
1007	34.97
1008	399.15

17. Use a query to show the invoices and invoice totals in Figure P7.17. Sort the results by customer code and then by invoice number.

FIGURE P7.17 INVOICE TOTALS BY CUSTOMER

CUS_CODE	INV_NUMBER	Invoice Total
10011	1002	9.96
10011	1004	34.97
10011	1008	399.15
10012	1003	153.95
10014	1001	24.94
10014	1006	397.83
10015	1007	34.97
10018	1005	70.44

18. Write a query to produce the number of invoices and the total purchase amounts by customer, using the output shown in Figure P7.18 as your guide. Note the results are sorted by customer code. (Compare this summary to the results shown in Problem 17.)

FIGURE P7.18 NUMBER OF INVOICES AND TOTAL PURCHASE AMOUNTS BY CUSTOMER

CUS_CODE	Number of Invoices	Total Customer Purchases
10011	3	444.00
10012	1	153.95
10014	2	422.77
10015	1	34.97
10018	1	70.44

19. Write a query to generate the total number of invoices, the invoice total for all of the invoices, the smallest of the customer purchase amounts, the largest of the customer purchase amounts, and the average of all the customer purchase amounts. Your output must match Figure P7.19.

FIGURE P7.19 NUMBER OF INVOICES, INVOICE TOTALS, MINIMUM, MAXIMUM, AND AVERAGE SALES

Total Invoices	Total Sales	Minimum Customer Purchases	Largest Customer Purchases	Average Customer Purchases
8	1126.03	34.97	444.00	225.21

20. List the balances of customers who have made purchases during the current invoice cycle—that is, for the customers who appear in the INVOICE table. The results of this query are shown in Figure P7.20, sorted by customer code.

FIGURE P7.20 BALANCES FOR CUSTOMERS WHO MADE PURCHASES

CUS_CODE	CUS_BALANCE
10011	0.00
10012	345.86
10014	0.00
10015	0.00
10018	216.55

21. Provide a summary of customer balance characteristics for customers who made purchases. Include the minimum balance, maximum balance, and average balance, as shown in Figure P7.21.

FIGURE P7.21 BALANCE SUMMARY FOR CUSTOMERS WHO MADE PURCHASES

Minimum Balance	Maximum Balance	Average Balance
0	345.86	112.48

22. Create a query to find the balance characteristics for all customers, including the total of the outstanding balances. The results of this query are shown in Figure P7.22.

FIGURE P7.22 BALANCE SUMMARY FOR ALL CUSTOMERS

Total Balances	Minimum Balance	Maximum Balance	Average Balance
2089.28	0.00	768.93	208.93

23. Find the listing of customers who did not make purchases during the invoicing period. Sort the results by customer code. Your output must match the output shown in Figure P7.23.

FIGURE P7.23 BALANCES OF CUSTOMERS WHO DID NOT MAKE PURCHASES

CUS_CODE	CUS_BALANCE
10010	0.00
10013	536.75
10016	221.19
10017	768.93
10019	0.00

24. Find the customer balance summary for all customers who have not made purchases during the current invoicing period. The results are shown in Figure P7.24.

FIGURE P7.24 SUMMARY OF CUSTOMER BALANCES FOR CUSTOMERS WHO DID NOT MAKE PURCHASES

Total Balance	Minimum Balance	Maximum Balance	Average Balance
1526.87	0.00	768.93	305.37

25. Create a query that summarizes the value of products currently in inventory. Note that the value of each product is a result of multiplying the units currently in inventory by the unit price. Sort the results in descending order by subtotal, as shown in Figure P7.25.

FIGURE P7.25 VALUE OF PRODUCTS CURRENTLY IN INVENTORY

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

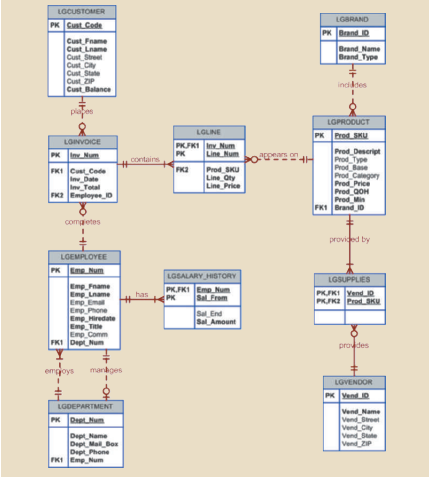
26. Find the total value of the product inventory. The results are shown in Figure P7.26.

FIGURE P7.26 TOTAL VALUE OF ALL PRODUCTS IN INVENTORY

Total Value of Inventory
15084.52

The Ch07_LargeCo database (see Figure P7.27) stores data for a company that sells paint products. The company tracks the sale of products to customers. The database keeps data on customers (LGCUSTOMER), sales (LGINVOICE), products (LGPRODUCT), which products are on which invoices (LGLINE), employees (LGEMPLOYEE), the salary history of each employee (LGSALARY_HISTORY), departments (LGDEPARTMENT), product brands (LGBRAND), vendors (LGVENDOR), and which vendors supply each product (LGSUPPLIES). Some of the tables contain only a few rows of data, while other tables are quite large; for example, there are only eight departments, but more than 3,300 invoices containing over 11,000 invoice lines. For Problems 28–55, a figure of the correct output for each problem is provided. If the output of the query is very large, only the first several rows of the output are shown.

FIGURE P7.27 THE CH07_LARGECO ERD



27. Write a query to display the eight departments in the LGDEPARTMENT table sorted by department name.

28. Write a query to display the SKU (stock keeping unit), description, type, base, category, and price for all products that have a PROD_BASE of Water and a PROD_CATEGORY of Sealer (Figure P7.28).

FIGURE P7.28 WATER-BASED SEALERS

PROD_SKU	PROD_DESCRIPT	PROD_TYPE	PROD_BASE	PROD_CATEGORY	PROD_PRICE
1403-TUY	Sealer, Water Based, for Concrete Floors	Interior	Water	Sealer	42.99

29. Write a query to display the first name, last name, and email address of employees hired from January 1, 2005, to December 31, 2014. Sort the output by last name and then by first name (Figure P7.29).

FIGURE P7.29 EMPLOYEES HIRED FROM 2005–2014

EMP_FNAME	EMP_LNAME	EMP_EMAIL
SUE	ASH	S.ASH99@LGCOMPANY.COM
ALIDA	BLACKWELL	A.BLACKWE99@LGCOMPANY.COM
PHOEBE	BLEDSOE	P.BLEDSO99@LGCOMPANY.COM
VALARIE	BLEDSOE	V.BLEDSO99@LGCOMPANY.COM
WILFORD	BURGOS	W.BURGOS9@LGCOMPANY.COM
KASEY	CASH	K.CASH@LGCOMPANY.COM
DANICA	CASTLE	C.DANICA99@LGCOMPANY.COM
DOUG	CAUDILL	C.DOUG@LGCOMPANY.COM
LUCIO	CAUDILL	L.CAUDILL@LGCOMPANY.COM
HANNAH	COLEMAN	H.COLEMA7@LGCOMPANY.COM
PHILLIS	CONKLIN	P.CONKLI@LGCOMPANY.COM
LEE	CONNOR	L.CONNOR99@LGCOMPANY.COM

30. Write a query to display the first name, last name, phone number, title, and department number of employees who work in department 300 or have the title "CLERK I." Sort the output by last name and then by first name (Figure P7.30).

FIGURE P7.30 CLERKS AND EMPLOYEES IN DEPARTMENT 300

EMP_FNAME	EMP_LNAME	EMP_PHONE	EMP_TITLE	DEPT_NUM
LAVINA	ACEVEDO	863-6797	ASSOCIATE	300
LAUREN	AVERY	668-2276	SENIOR ASSOCIATE	300
ROSALBA	BAKER	632-8197	ASSOCIATE	300
FERN	CARPENTER	736-4628	PURCHASING SPECIALIST	300
LEEANN	CLINTON	616-9615	CLERK I	600
TANIKA	CRANE	449-6336	PURCHASING SPECIALIST	300
SAMMY	DIGGS	526-2101	SENIOR ASSOCIATE	300
LANA	DOWDY	471-8796	SENIOR ASSOCIATE	300
STEPHANIE	DUNLAP	918-9203	BUYER - RAW MATERIALS	300
HAL	FISHER	676-3662	SENIOR ASSOCIATE	300
LINDSAY	GOOD	337-9578	CLERK I	600
LEEANN	HORN	528-4361	SENIOR ASSOCIATE	300

31. Write a query to display the employee number, last name, first name, salary "from" date, salary end date, and salary amount for employees 83731, 83745, and 84039. Sort the output by employee number and salary "from" date (Figure P7.31).

FIGURE P7.31 SALARY HISTORY FOR SELECTED EMPLOYEES

EMP_NUM	EMP_LNAME	EMP_FNAME	SAL_FROM	SAL_END	SAL_AMOUNT
83731	VARGAS	SHERON	7/15/2014	7/14/2015	43740
83731	VARGAS	SHERON	7/14/2015	7/13/2016	48110
83731	VARGAS	SHERON	7/14/2016	7/14/2017	49550
83731	VARGAS	SHERON	7/15/2017		51040
83745	SPICER	DWAIN	8/2/2011	8/1/2012	56020
83745	SPICER	DWAIN	8/2/2012	8/2/2013	57700
83745	SPICER	DWAIN	8/3/2013	8/1/2014	63470
83745	SPICER	DWAIN	8/2/2014	8/1/2015	68550
83745	SPICER	DWAIN	8/1/2015	7/31/2016	71980
83745	SPICER	DWAIN	8/1/2016	8/1/2017	74140
83745	SPICER	DWAIN	8/2/2017		76360
84039	COLEMAN	HANNAH	6/28/2014	6/27/2015	47380
84039	COLEMAN	HANNAH	6/27/2015	6/26/2016	51170
84039	COLEMAN	HANNAH	6/27/2016	6/27/2017	52700
84039	COLEMAN	HANNAH	6/28/2017		54280

32. Write a query to display the first name, last name, street, city, state, and zip code of any customer who purchased a Foresters Best brand top coat between July 15, 2015, and July 31, 2015. If a customer purchased more than one such product, display the customer's information only once in the output. Sort the output by state, last name, and then first name (Figure P7.32).

FIGURE P7.32 CUSTOMERS WHO PURCHASED FORESTERS BEST TOP COAT

CUST_FNAME	CUST_LNAME	CUST_STREET	CUST_CITY	CUST_STATE	CUST_ZIP
LUPE	SANTANA	1290 WEST 70TH PLACE	Phenix City	AL	36867
HOLLIS	STILES	1493 DOLLY MADISON CIRCLE	Snow Hill	AL	36778
LISETTE	WHITTAKER	339 NORTHPARK DRIVE	Montgomery	AL	36197
DEANDRE	JAMISON	1571 HANES STREET	Miami	FL	33169
CATHLEEN	WHITMAN	1712 NORTHFIELD DRIVE	Marshallville	GA	31057
SHERIE	STOVER	640 MOUNTAIN VIEW DRIVE	Parkville	KY	40464
BRYCE	HOGAN	1880 IMLACH DRIVE	Newbury	MA	01951
SHELBY	SALAS	486 SUSITNA VIEW COURT	North Tisbury	MA	02568
JERMAINE	HANCOCK	1627 SAUNDERS ROAD	Ellicott City	MD	21041
WHITNEY	WHITFIELD	1259 RHONE STREET	Phippsburg	ME	04567
MONROE	ALLISON	273 SCHODDE STREET	Kalamazoo	MI	49002
DARLEEN	PARRA	561 COLLIE HILL WAY	Madison	MS	39130
CLINTON	AGUIRRE	1651 VANGUARD DRIVE	Franklinville	NC	27248
TOMMIE	PALMER	933 ELCADORE CIRCLE	Arapahoe	NC	28510
JEFFEREY	MCBRIDE	1043 ROCKRIDGE DRIVE	Glenwood	NJ	07418
SIDNEY	GARZA	772 SHEPPARD DRIVE	Fair Harbor	NY	11706
TAMELA	GUIDRY	1873 BAXTER ROAD	Brooklyn	NY	11252
KAREN	LEVINE	1534 PALMER COURT	Cincinnati	OH	45218
STEPHENIE	MCKENZIE	1039 DELAWARE PLACE	Wilkes Barre	PA	18763
LAN	NICHOLS	367 LAKEVIEW DRIVE	Pittsburgh	PA	15262
KASEY	SOSA	975 WEST 56TH AVENUE	Kinzers	PA	17535
SHELBY	THAYER	1634 RUANE ROAD	Bordeaux	SC	29935
WILSON	BELL	1127 CUNNINGHAM STREET	Louisville	TN	37777
RENATE	LADD	652 LEWIS STREET	Crystal City	VA	22202
MELONIE	JIMENEZ	848 DOWNEY FINCH LANE	East Monkton	VT	05443

33. Write a query to display the employee number, last name, email address, title, and department name of each employee whose job title ends in the word "ASSOCIATE." Sort the output by department name and employee title (Figure P7.33).

FIGURE P7.33 EMPLOYEE WITH THE TITLE OF ASSOCIATE

EMP_NUM	EMP_LNAME	EMP_EMAIL	EMP_TITLE	DEPT_NAME
84636	LASSITER	F.LASSITE@LGCOMPANY.COM	ASSOCIATE	ACCOUNTING
83617	ALBRIGHT	SO.ALBRIG6@LGCOMPANY.COM	ASSOCIATE	ACCOUNTING
84366	RIVERA	D.RIVERA76@LGCOMPANY.COM	ASSOCIATE	ACCOUNTING
83378	DUNHAM	F.DUNHAM5@LGCOMPANY.COM	ASSOCIATE	ACCOUNTING
83583	ROLLINS	M.ROLLINS@LGCOMPANY.COM	ASSOCIATE	ACCOUNTING
83961	FINN	D.FINN7@LGCOMPANY.COM	ASSOCIATE	ACCOUNTING
84383	WASHINGTON	L.WASHINGT@LGCOMPANY.COM	ASSOCIATE	CUSTOMER SERVICE
84206	HEALY	N.HEALY0@LGCOMPANY.COM	ASSOCIATE	CUSTOMER SERVICE
83451	ELLIS	R.ELLIS9@LGCOMPANY.COM	ASSOCIATE	CUSTOMER SERVICE
84442	GREGORY	A.GREGORY8@LGCOMPANY.COM	ASSOCIATE	CUSTOMER SERVICE
84459	GILLIAM	E.GILLIA10@LGCOMPANY.COM	ASSOCIATE	CUSTOMER SERVICE
84300	SEAY	A.SEAY75@LGCOMPANY.COM	ASSOCIATE	CUSTOMER SERVICE

34. Write a query to display a brand name and the number of products of that brand that are in the database. Sort the output by the brand name (Figure P7.34).

FIGURE P7.34 NUMBER OF PRODUCTS OF EACH BRAND

BRAND_NAME	NUMPRODUCTS
BINDER PRIME	27
BUSTERS	25
FORESTERS BEST	15
HOME COMFORT	36
LE MODE	36
LONG HAUL	41
OLDE TYME QUALITY	27
STUTTENFURST	27
VALU-MATTE	18

35. Write a query to display the number of products in each category that have a water base, sorted by category (Figure P7.35).

FIGURE P7.35 NUMBER OF WATER-BASED PRODUCTS IN EACH CATEGORY

PROD_CATEGORY	NUMPRODUCTS
Cleaner	2
Filler	2
Primer	16
Sealer	1
Top Coat	81

36. Write a query to display the number of products within each base and type combination, sorted by base and then by type (Figure P7.36).

FIGURE P7.36 NUMBER OF PRODUCTS OF EACH BASE AND TYPE

PROD_BASE	PROD_TYPE	NUMPRODUCTS
Solvent	Exterior	67
Solvent	Interior	83
Water	Exterior	39
Water	Interior	63

37. Write a query to display the total inventory—that is, the sum of all products on hand for each brand ID. Sort the output by brand ID in descending order (Figure P7.37).

FIGURE P7.37 TOTAL INVENTORY OF EACH BRAND OF PRODUCTS

BRAND_ID	TOTALINVENTORY
35	2431
33	2158
31	1117
30	3012
29	1735
28	2200
27	2586
25	1829
23	1293

38. Write a query to display the brand ID, brand name, and average price of products of each brand. Sort the output by brand name. Results are shown with the average price rounded to two decimal places (Figure P7.38).

FIGURE P7.38 AVERAGE PRICE OF PRODUCTS OF EACH BRAND

BRAND_ID	BRAND_NAME	AVGPRICE
33	BINDER PRIME	16.12
29	BUSTERS	22.69
23	FORESTERS BEST	20.94
27	HOME COMFORT	21.6
35	LE MODE	19.22
30	LONG HAUL	20.12
28	OLDE TYME QUALITY	18.33
25	STUTTENFURST	16.47
31	VALU-MATTE	16.84

39. Write a query to display the department number and most recent employee hire date for each department. Sort the output by department number (Figure P7.39).

FIGURE P7.39 MOST RECENT HIRE IN EACH DEPARTMENT

DEPT_NUM	MOSTRECENT
200	6/8/2007
250	12/15/2017
260	4/16/2016
300	12/12/2016
400	1/26/2017
500	4/26/2017
550	10/22/2017
600	10/2/2017

40. Write a query to display the employee number, first name, last name, and largest salary amount for each employee in department 200. Sort the output by largest salary in descending order (Figure P7.40).

FIGURE P7.40 LARGEST SALARY AMOUNT FOR EACH EMPLOYEE IN DEPARTMENT 200

EMP_NUM	EMP_FNAME	EMP_LNAME	LARGESTSALARY
83699	FRANKLYN	STOVER	210000
83795	JODIE	BARR	147000
83537	CLEO	ENGLISH	136000
83565	LOURDES	ABERNATHY	133000
83693	ROSANNE	NASH	129000
83621	FONDA	GONZALEZ	126000
83649	DELMA	JACOB	123000
83677	HERB	MANNING	120000
83936	BRADFORD	BRAY	117000
83734	INEZ	ROCHA	112000
84049	LANE	BRANDON	110000
83763	JAIME	FELTON	107000

41. Write a query to display the customer code, first name, last name, and sum of all invoice totals for customers with cumulative invoice totals greater than $1,500. Sort the output by the sum of invoice totals in descending order (Figure P7.41).

FIGURE P7.41 SUM OF CUSTOMER PURCHASES FOR CUSTOMERS PURCHASING MORE THAN $1500

CUST_CODE	CUST_FNAME	CUST_LNAME	TOTALINVOICES
795	CHARMAINE	BRYAN	3134.16
98	VALENTIN	MARINO	3062.46
152	LISETTE	WHITTAKER	3042.78
157	KARON	MATA	3009.63
97	ERWIN	ANDERSON	2895.49
112	LAN	NICHOLS	2867.14
158	JESSE	HICKS	2786.55
220	ABRAHAM	PLATT	2187.36
103	CORRINA	GIFFORD	2122.07
362	SHIRLENE	FITCH	2046.31
173	INGRID	HARDY	2040.31
132	JAMIS	DUBOIS	2015.62

42. Write a query to display the department number, department name, department phone number, employee number, and last name of each department manager. Sort the output by department name (Figure P7.42).

FIGURE P7.42 DEPARTMENT MANAGERS

DEPT_NUM	DEPT_NAME	DEPT_PHONE	EMP_NUM	EMP_LNAME
600	ACCOUNTING	555-2333	84993	YAZDE
250	CUSTOMER SERVICE	555-5555	84001	FARMER
600	DISTRIBUTION	555-3624	84052	FORD
200	MARKETING	555-9500	84042	PETTIT
300	PURCHASING	555-4873	83746	RANKIN
200	SALES	555-2824	83509	STOVER
550	TRUCKING	555-3057	83883	STONE
400	WAREHOUSE	555-1063	83759	CHARLES

43. Write a query to display the vendor ID, vendor name, brand name, and number of products of each brand supplied by each vendor. Sort the output by vendor name and then by brand name (Figure P7.43).

FIGURE P7.43 NUMBER OF PRODUCTS OF EACH BRAND SUPPLIED BY EACH VENDOR

VEND_ID	VEND_NAME	BRAND_NAME	NUMPRODUCTS
8	Baltimore Paints Consolidated	BINDER PRIME	27
8	Baltimore Paints Consolidated	FORESTERS BEST	1
8	Baltimore Paints Consolidated	HOME COMFORT	36
8	Baltimore Paints Consolidated	LE MODE	3
8	Baltimore Paints Consolidated	LONG HAUL	3
8	Baltimore Paints Consolidated	VALU-MATTE	1
13	Boykin Chemical Workshop	BUSTERS	1
13	Boykin Chemical Workshop	LE MODE	2
13	Boykin Chemical Workshop	LONG HAUL	2
13	Boykin Chemical Workshop	OLDE TYME QUALITY	2
13	Boykin Chemical Workshop	STUTTERFURST	1
13	Boykin Chemical Workshop	VALU-MATTE	1

44. Write a query to display the employee number, last name, first name, and sum of invoice totals for all employees who completed an invoice. Sort the output by employee last name and then by first name (Figure P7.44).

FIGURE P7.44 TOTAL VALUE OF INVOICES COMPLETED BY EACH EMPLOYEE

EMP_NUM	EMP_LNAME	EMP_FNAME	TOTALINVOICES
83665	ABERNATHY	LOURDES	19158.54
83792	ANDERSEN	WALLY	20627.47
83795	BARR	JOSE	23099.88
84049	BRANDOIN	LANE	20863.06
83935	BRAY	BRADFORD	21139.94
84248	CASTLE	DANICA	17760.42
84430	CAUDILL	DOUG	11308.21
83893	CORTEZ	SANG	17436.88
84021	DICHINSON	LARCO	20437.35
84153	EASLEY	GWEN	24013.26
83537	ENGLISH	CLEO	18883.13
84078	ERWIN	DIEGO	23839.85

45. Write a query to display the largest average product price of any brand (Figure P7.45).

FIGURE P7.45 LARGEST AVERAGE BRAND PRICE

LARGEST AVERAGE
22.59

46. Write a query to display the brand ID, brand name, brand type, and average price of products for the brand that has the largest average product price (Figure P7.46).

FIGURE P7.46 BRAND WITH THE HIGEST AVERAGE PRICE

BRAND_ID	BRAND_NAME	BRAND_TYPE	AVGPRICE
29	BUSTERS	VALUE	22.59

47. Write a query to display the manager name, department name, department phone number, employee name, customer name, invoice date, and invoice total for the department manager of the employee who made a sale to a customer whose last name is Hagan on May 18, 2015 (Figure P7.47).

FIGURE P7.47 MANAGER OF EMPLOYEE MAKING A SALE TO CUSTOMER HAGAN

Manager FName	Manager LName	DEPT_NAME	DEPT_PHONE	Employee Fname	Employee Lname	Customer FName	Customer LName	INV_DATE	INV_TOTAL
FRANKLYN	STOVER	SALES	555-2824	THURMAN	WILKINSON	DARELL	HAGAN	5/18/2015	316.04

48. Write a query to display the current salary for each employee in department 300. Assume that only current employees are kept in the system, and therefore the most current salary for each employee is the entry in the salary history with a NULL end date. Sort the output in descending order by salary amount (Figure P7.48).

FIGURE P7.48 CURRENT SALARY FOR EMPLOYEES IN DEPARTMENT 300

Emp_Num	Emp_LName	Emp_FName	Sal_Amount
83745	RANKIN	SEAN	96550
84328	CARPENTER	FERN	94090
83716	RIVERA	HENRY	85920
84432	JAMISON	MERLE	85360
83902	VARGAS	ROCKY	79540
83695	MENDEZ	CARROLL	75200
84500	WESTON	CHRISTINE	78690
84694	TIDWELL	ODELL	77480
83910	AVERY	LAUREN	76110
83359	WATTS	MERLE	72240
83780	ACEVEDO	LAVINIA	72000

49. Write a query to display the starting salary for each employee. The starting salary would be the entry in the salary history with the oldest salary start date for each employee. Sort the output by employee number (Figure P7.49).

FIGURE P7.49 STARTING SALARY FOR EACH EMPLOYEE

Emp_Num	Emp_LName	Emp_FName	Sal_Amount
83304	MCDONALD	TAMARA	19770
83308	LOVE	CONNIE	11200
83312	BAKER	ROSALBA	39200
83314	DAVID	CHAROLETTE	15150
83318	PECK	DARCIE	22330
83321	FARMER	ANGELINA	18250
83322	LONG	WILLARD	23360
83341	CORTEZ	CHRISTINE	14510
83347	WINN	QUINTIN	17010
83349	SINGH	JENNIFFER	21220
83359	WATTS	MERLE	25370
83366	BLEDSOE	PHOEBE	23200

50. Write a query to display the invoice number, line numbers, product SKUs, product descriptions, and brand ID for sales of sealer and top coat products of the same brand on the same invoice. Sort the results by invoice number in ascending order, first line number in ascending order, and then by second line number in descending order (Figure P7.50).

FIGURE P7.50 INVOICES FOR SEALER AND TOP COAT OF THE SAME BRAND

Inv_Num	Line_Num	p Prod_Sku	p Prod_Descript	D Line_Num	p2 Prod_Sku	p2 Prod_Descript	Brand_ID
115	2	5140-RTG	Fire Resistant Sealer, for Exterior Wood (ULC Approved)	1	1320-AIS	Fire Retardant Coating, Latex, Interior, Flat (ULC Approved)	36
119	2	5140-RTG	Fire Resistant Sealer, for Exterior Wood (ULC Approved)	5	5566-TTC	Aluminum Paint, Heat Resistant (Up to 427°C - 800°F)	36
135	5	3030-PCT	Sealer, for Knots	2	1974-VVJ	Light Industrial Coating, Exterior, Water Based (Eggshell-like) - MPI Gloss Level 3	25
163	2	2701-YAW	Sealer, Solvent Based, for Concrete Floors	1	3956-WWD	Water Repellant, Clear Wet Formula	36
222	1	1336-FVM	Alkyd, Sanding Sealer, Clear	3	8199-YRF	Varnish, Exterior, Water Based, (Satin-Like) MPI Gloss Level 4	08
234	4	5720-ZPO	Shop Coat, Quick Dry, for Interior Steel	3	9272-LTP	Varnish, Marine Spar, Exterior, Gloss (MPI Gloss Level 6)	27
234	4	5720-ZPO	Shop Coat, Quick Dry, for Interior Steel	2	9126-PWF	Latex, Recycled (Consolidated), Interior (MPI Gloss Level 2)	27
240	1	4072-SWV	Sealer, Solvent Based, for Concrete Floors	3	5803-RTU	Aluminum Paint	25
287	1	8854-LUR	Lacquer, Sanding Sealer, Clear	5	9835-FUF	Fire Retardant Top-Coat, Clear, Alkyd, Interior (ULC Approved)	37
332	1	3701-YAW	Sealer, Solvent Based, for Concrete Floors	6	2884-GU	Stain, for Exterior Wood Decks	36
333	1	3701-YAW	Sealer, Solvent Based, for Concrete Floors	5	4784-SLU	Lacquer, Clear, Flat	36
369	3	1403-TUY	Sealer, Water Based, for Concrete Floors	1	9736-ZNM	Floor Paint, Alkyd, Low Gloss	29

51. The Binder Prime Company wants to recognize the employee who sold the most of its products during a specified period. Write a query to display the employee number, employee first name, employee last name, email address, and total units sold for the employee who sold the most Binder Prime brand products between November 1, 2015, and December 5, 2015. If there is a tie for most units sold, sort the output by employee last name (Figure P7.51).

FIGURE P7.49 STARTING SALARY FOR EACH EMPLOYEE

Emp_Num	Emp_FName	Emp_LName	Emp_Email	Total
84134	ROSALIE	GARLAND	G.ROSALIE@LGCOMPANY.COM	23
83950	RUSTY	MILES	M.RUSTY95@LGCOMPANY.COM	23

52. Write a query to display the customer code, first name, and last name of all customers who have had at least one invoice completed by employee 83649 and at least one invoice completed by employee 83677. Sort the output by customer last name and then first name (Figure P7.52).

FIGURE P7.52 CUSTOMERS WITH INVOICE FILLED BY EMPLOYEES 83649 AND 83677

Cust_Code	Cust_FName	Cust_LName
684	WENDI	BEAN
340	MARCIA	BURRIS
211	GERALD	CAUDILL
292	VALARIE	DILLARD
293	CLAIR	ERICKSON
416	TATIANA	HOWE
996	EZRA	LYON
90	VALENTIN	MARINO
121	PETER	SMALL
1167	LUCIO	STALEY
617	CESAR	TALLEY
457	SHAUNA	WERNER
131	SAL	WHALEY

53. LargeCo is planning a new promotion in Alabama (AL) and wants to know about the largest purchases made by customers in that state. Write a query to display the customer code, customer first name, last name, full address, invoice date, and invoice total of the largest purchase made by each customer in Alabama. Be certain to include any customers in Alabama who have never made a purchase; their invoice dates should be NULL and the invoice totals should display as 0. Sort the results by customer last name and then first name (Figure P7.53).

FIGURE P7.53 LARGEST PURCHASES OF CUSTOMERS IN AL

Cust_Code	Cust_FName	Cust_LName	Cust_Street	Cust_City	Cust_State	Cust_ZIP	Inv_Date	Largest Invoice
903	ROBIN	ADDISON	323 LORETTA PLACE	Mobile	AL	36693	8/26/2015	230.63
643	NINA	ALLEN	680 RED TALON DRIVE	Robertsdale	AL	36574	6/21/2015	11.99
295	DORTHY	AUSTIN	829 BIG BEND LOOP	Diamond Shamrock	AL	36614	4/24/2015	589.75
393	FOSTER	BERNAL	1299 EAST 3RD AVENUE	Birmingham	AL	35280		0
853	GAYLORD	BOLTON	1069 LUGENE LANE	Montgomery	AL	36131	11/25/2015	372.68
925	ALANA	BOOKER	1874 I STREET	Mccullough	AL	36502	12/12/2015	208.85
1248	LISA	BRADY	491 LOWLAND AVENUE	Daphne	AL	36577	12/5/2015	414.47
538	CHIQUITA	CALDWELL	1501 BRIGGS COURT	Normal	AL	35762	5/26/2015	143.9
89	MONICA	CANTRELL	697 ADAK CIRCLE	Loachapoka	AL	36865	3/31/2015	516.59
1233	NATHALIE	CHURCH	1802 SNOWY OWL CIRCLE	Napier Field	AL	36303	11/24/2015	160.96
304	GERTRUDE	CONNORS	1842 PLEASANT DRIVE	Georgiana	AL	36033	12/29/2015	376.32
1131	CARMA	CORNETT	767 CHISANA WAY	Killen	AL	35645	10/25/2015	265.12
1407	FELICIA	CRUZ	643 TURNAGAIN PARKWAY	Coalburg	AL	35068	1/6/2016	387.93

54. One of the purchasing managers is interested in the impact of product prices on the sale of products of each brand. Write a query to display the brand name, brand type, average price of products of each brand, and total units sold of products of each brand. Even if a product has been sold more than once, its price should only be included once in the calculation of the average price. However, you must be careful because multiple products of the same brand can have the same price, and each of those products must be included in the calculation of the brand's average price. Sort the result by brand name (Figure P7.54).

FIGURE P7.54 AVERAGE PRICE AND TOTAL UNITS SOLD OF PRODUCTS BY BRAND

Brand_Name	Brand_Type	Average Price	Units Sold
BINDER PRIME	PREMIUM	16.12	3753
BUSTERS	VALUE	22.59	3727
FORESTERS BEST	VALUE	20.94	2086
HOME COMFORT	CONTRACTOR	21.8	4042
LE MODE	PREMIUM	19.22	5284
LONG HAUL	CONTRACTOR	20.12	5739
OLDE TYME QUALITY	CONTRACTOR	18.33	3614
STUTTENFURST	CONTRACTOR	16.47	3671
VALU-MATTE	VALUE	16.84	2495

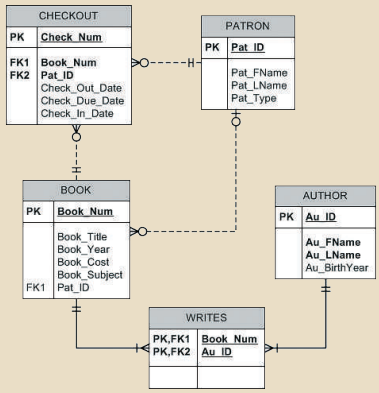
55. The purchasing manager is still concerned about the impact of price on sales. Write a query to display the brand name, brand type, product SKU, product description, and price of any products that are not a premium brand, but that cost more than the most expensive premium brand products (Figure P7.55).

FIGURE P7.55 NON-PREMIUM PRODUCTS THAT ARE MORE EXPENSIVE THAN PREMIUM PRODUCTS

Brand_Name	Brand_Type	Prod_Sku	Prod_Descript	Prod_Price
LONG HAUL	CONTRACTOR	1964-OUT	Fire Resistant Top Coat, for Interior Wood	78.49

The CIS Department at Tiny College maintains the Free Access to Current Technology (FACT) library of e-books. FACT is a collection of current technology e-books for use by faculty and students. Agreements with the publishers allow patrons to electronically check out a book, which gives them exclusive access to the book online through the FACT website, but only one patron at a time can have access to a book. A book must have at least one author but can have many. An author must have written at least one book to be included in the system but may have written many. A book may have never been checked out but can be checked out many times by the same patron or different patrons over time. Because all faculty and staff in the department are given accounts at the online library, a patron may have never checked out a book or they may have checked out many books over time. To simplify determining which patron currently has a given book checked out, a redundant relationship between BOOK and PATRON is maintained. The ERD for this system is shown in Figure P7.56 and should be used to answer the next several problems. For Problems 57–109, a figure of the correct output is provided for each problem. If the output of the query is very large, only the first several rows of the output are shown.

FIGURE P7.56 THE CH07_FACT ERD



56. Write a query that displays the book title, cost and year of publication for every book in the system. Sort the results by book title.

57. Write a query that displays the first and last name of every patron, sorted by last name and then first name. Ensure the sort is case insensitive (Figure P7.57). (50 rows)

FIGURE P7.57 ALL PATRON NAMES

PAT_FNAME	PAT_LNAME
Vera	Alvarado
Holly	Anthony
Cedric	Baldwin
Cory	Barry
Nadine	Blair
Erika	Bowen
Gerald	Burke
Ollie	Cantrell
robert	carter
Keith	Cooley

58. Write a query to display the checkout number, checkout date, and due date for every book that has been checked out sorted by checkout number (Figure P7.58). (68 rows)

FIGURE P7.58 ALL CHECKOUTS

CHECK_NUM	CHECK_OUT_DATE	CHECK_DUE_DATE
91001	3/31/2017	4/14/2017
91002	3/31/2017	4/7/2017
91003	3/31/2017	4/14/2017
91004	3/31/2017	4/14/2017
91005	3/31/2017	4/7/2017
91006	4/5/2017	4/12/2017
91007	4/5/2017	4/12/2017
91008	4/5/2017	4/12/2017
91009	4/5/2017	4/19/2017
91010	4/5/2017	4/19/2017
91011	4/5/2017	4/12/2017

59. Write a query to display the book number, book title, and subject for every book sorted by book number (Figure P7.59). (20 rows)

FIGURE P7.59 TITLE AND SUBJECT FOR ALL BOOKS

BOOK_NUM	TITLE	Subject of Book
5235	Beginner's Guide to JAVA	Programming
5236	Database in the Cloud	Cloud
5237	Mastering the database environment	Database
5238	Conceptual Programming	Programming
5239	J++ in Mobile Apps	Programming
5240	iOS Programming	Programming
5241	JAVA First Steps	Programming
5242	C# in Middleware Deployment	Middleware
5243	DATABASES in Theory	Database
5244	Cloud-based Mobile Applications	Cloud
5245	The Golden Road to Platform independence	Middleware

60. Write a query to display the different years in which books have been published. Include each year only once and sort the results by year (Figure P7.60).

FIGURE P7.60 UNIQUE BOOK YEARS

BOOK_YEAR
2014
2015
2016
2017

61. Write a query to display the different subjects on which FACT has books. Include each subject only once and sort the results by subject (Figure P7.61).

FIGURE P7.61 UNIQUE BOOK SUBJECTS

BOOK_SUBJECT
Cloud
Database
Middleware
Programming

62. Write a query to display the book number, title, and cost of each book sorted by book number (Figure P7.62).

FIGURE P7.62 TITLE AND REPLACEMENT COST FOR BOOKS

BOOK_NUM	BOOK_TITLE	Replacement Cost
5235	Beginner's Guide to JAVA	59.95
5236	Database in the Cloud	79.95
5237	Mastering the database environment	69.95
5238	Conceptual Programming	59.95
5239	J++ in Mobile Apps	49.95
5240	iOS Programming	79.95
5241	JAVA First Steps	49.95
5242	C# in Middleware Deployment	59.95
5243	DATABASES in Theory	129.95
5244	Cloud-based Mobile Applications	69.95
5245	The Golden Road to Platform independence	119.95
5246	Capture the Cloud	69.95
5247	Shining Through the Cloud: Sun Programming	109.95
5248	What You Always Wanted to Know About Database, But Were Afraid to Ask	49.95

63. Write a query to display the checkout number, book number, patron ID, checkout date, and due date for every checkout that has ever occurred in the system. Sort the results by checkout date in descending order and then by checkout number in ascending order (Figure P7.63). (68 rows)

FIGURE P7.63 CHECKOUTS BY DATE

CHECK_NUM	BOOK_NUM	PAT_ID	CHECK_OUT_DATE	CHECK_DUE_DATE
91067	5252	1229	5/24/2017	5/31/2017
91068	5238	1229	5/24/2017	5/31/2017
91066	5242	1228	5/19/2017	5/26/2017
91064	5236	1183	5/17/2017	5/31/2017
91065	5244	1210	5/17/2017	5/24/2017
91060	5235	1209	5/15/2017	5/22/2017
91061	5246	1172	5/15/2017	5/22/2017
91062	5254	1223	5/15/2017	5/22/2017
91063	5243	1223	5/15/2017	5/22/2017
91056	5254	1224	5/10/2017	5/17/2017

64. Write a query to display the book title, year, and subject for every book. Sort the results by book subject in ascending order, year in descending order, and then title in ascending order (Figure P7.64). (20 rows)

FIGURE P7.64 BOOKS BY CASCADING SORT

BOOK_TITLE	BOOK_YEAR	BOOK_SUBJECT
Capture the Cloud	2016	Cloud
Starlight Applications	2016	Cloud
Cloud-based Mobile Applications	2015	Cloud
Database in the Cloud	2014	Cloud
Beyond the Database Veil	2016	Database
What You Always Wanted to Know About Database, But Were Afraid to Ask	2016	Database
DATABASES in Theory	2015	Database
Mastering the database environment	2015	Database
Reengineering the Middle Tier	2016	Middleware
The Golden Road to Platform independence	2016	Middleware

65. Write a query to display the book number, title, and cost for all books that cost $59.95 sorted by book number (Figure P7.65).

FIGURE P7.65 BOOKS THAT COST $59.95

BOOK_NUM	BOOK_TITLE	BOOK_COST
5235	Beginner's Guide to JAVA	59.95
5238	Conceptual Programming	59.95
5242	C# in Middleware Deployment	59.95
5251	Thoughts on Revitalizing Ruby	59.95

66. Write a query to display the book number, title, and replacement cost for all books in the "Database" subject sorted by book number (Figure P7.66).

FIGURE P7.66 DATABASE BOOKS

BOOK_NUM	BOOK_TITLE	BOOK_COST
5237	Mastering the database environment	89.95
5243	DATABASES in Theory	129.95
5248	What You Always Wanted to Know About Database, But Were Afraid to Ask	49.95
5252	Beyond the Database Veil	69.95

67. Write a query to display the checkout number, book number, and checkout date of all books checked out before April 5, 2017 sorted by checkout number (Figure P7.67).

FIGURE P7.67 CHECKOUTS BEFORE APRIL 5TH

CHECK_NUM	BOOK_NUM	CHECK_OUT_DATE
91001	5235	3/31/2017
91002	5238	3/31/2017
91003	5240	3/31/2017
91004	5237	3/31/2017
91005	5236	3/31/2017

68. Write a query to display the book number, title, and year of all books published after 2015 and on the "Programming" subject sorted by book number (Figure P7.68).

FIGURE P7.68 NEWER BOOKS ON PROGRAMMING

BOOK_NUM	BOOK_TITLE	BOOK_YEAR
5247	Shining Through the Cloud: Sun Programming	2016
5251	Thoughts on Revitalizing Ruby	2016
5253	Virtual Programming for Virtual Environments	2016
5254	Coding Style for Maintenance	2017

69. Write a query to display the book number, title, subject, and cost for all books that are on the subjects of "Middleware" or "Cloud," and that cost more than $70 sorted by book number (Figure P7.69).

FIGURE P7.69 EXPENSIVE MIDDLEWARE OR CLOUD BOOKS

BOOK_NUM	BOOK_TITLE	BOOK_SUBJECT	BOOK_COST
5236	Database in the Cloud	Cloud	79.95
5245	The Golden Road to Platform independence	Middleware	119.95
5250	Reengineering the Middle Tier	Middleware	89.95

70. Write a query to display the author ID, first name, last name, and year of birth for all authors born in the decade of the 1980s sorted by author ID (Figure P7.70).

FIGURE P7.70 AUTHORS BORN IN THE 1980S

AU_ID	AU_FNAME	AU_LNAME	AU_BIRTHYEAR
218	Rachel	Beatney	1983
383	Neal	Walsh	1980
394	Robert	Lake	1982
438	Perry	Pearson	1986
460	Connie	Paulsen	1983
581	Manish	Aggerwal	1984
603	Julia	Palca	1988

71. Write a query to display the book number, title, and subject for all books that contain the word "Database" in the title, regardless of how it is capitalized. Sort the results by book number (Figure P7.71).

FIGURE P7.71 BOOK TITLES CONTAINING DATABASE

BOOK_NUM	BOOK_TITLE	BOOK_SUBJECT
5236	Database in the Cloud	Cloud
5237	Mastering the database environment	Database
5243	DATABASES in Theory	Database
5248	What You Always Wanted to Know About Database, But Were Afraid to Ask	Database
5252	Beyond the Database Veil	Database

72. Write a query to display the patron ID, first and last name of all patrons who are students, sorted by patron ID (Figure P7.72). (44 rows)

FIGURE P7.72 STUDENT PATRONS

PAT_ID	PAT_FNAME	PAT_LNAME
1166	Vera	Alvarado
1171	Peggy	Marsh
1172	Tony	Miles
1174	Betsy	Malone
1180	Nadine	Blair
1181	Allen	Horne
1182	Jamal	Melendez
1184	Jimmie	Love
1185	Sandra	Yang
1200	Lorenzo	Torres

73. Write a query to display the patron ID, first and last name, and patron type for all patrons whose last name begins with the letter "C," sorted by patron ID (Figure P7.73).

FIGURE P7.73 PATRONS WHOSE LAST NAME STARTS WITH "C"

PAT_ID	PAT_FNAME	PAT_LNAME	PAT_TYPE
1160	robert	carter	Faculty
1208	Ollie	Cantrell	Student
1210	Keith	Cooley	STUdent

74. Write a query to display the author ID, first and last name of all authors whose year of birth is unknown. Sort the results by author ID (Figure P7.74).

FIGURE P7.74 AUTHORS WITH UNKNOWN BIRTH YEAR

AU_ID	AU_FNAME	AU_LNAME
229	Carmine	Salvadore
262	Xia	Chiang
559	Rachel	McGill

75. Write a query to display the author ID, first and last name of all authors whose year of birth is known. Ensure the results are sorted by author ID (Figure P7.75).

FIGURE P7.75 AUTHORS WITH KNOWN BIRTH YEAR

AU_ID	AU_FNAME	AU_LNAME
185	Benson	Reeves
218	Rachel	Beatney
251	Hugo	Bruer
273	Reba	Durante
284	Trina	Tankersly
383	Neal	Walsh
394	Robert	Lake
438	Perry	Pearson
460	Connie	Paulsen
581	Manish	Aggerwal
592	Lawrence	Sheel
603	Julia	Palca

76. Write a query to display the checkout number, book number, patron ID, checkout date, and due date for all checkouts that have not yet been returned. Sort the results by book number (Figure P7.76).

FIGURE P7.76 UNRETURNED CHECKOUTS

CHECK_NUM	BOOK_NUM	PAT_ID	CHECK_OUT_DATE	CHECK_DUE_DATE
91068	5238	1229	5/24/2017	5/31/2017
91053	5240	1212	5/9/2017	5/16/2017
91066	5242	1228	5/19/2017	5/26/2017
91061	5246	1172	5/15/2017	5/22/2017
91059	5249	1207	5/10/2017	5/17/2017
91067	5252	1229	5/24/2017	5/31/2017

77. Write a query to display the author ID, first name, last name, and year of birth for all authors. Sort the results in descending order by year of birth, and then in ascending order by last name (Figure P7.77). (*Note:* Some DBMS sort NULLs as being large and some DBMS sort NULLs as being small.)

FIGURE P7.77 AUTHORS BY BIRTH YEAR

AU_ID	AU_FNAME	AU_LNAME	AU_BIRTHYEAR
185	Benson	Reeves	1990
603	Julia	Palca	1988
438	Perry	Pearson	1986
581	Manish	Aggerwal	1984
218	Rachel	Beatney	1983
460	Connie	Paulsen	1983
394	Robert	Lake	1982
383	Neal	Walsh	1980
592	Lawrence	Sheel	1976
251	Hugo	Bruer	1972
273	Reba	Durante	1969
284	Trina	Tankersly	1961
262	Xia	Chiang	
559	Rachel	McGill	
229	Carmine	Salvadore	

78. Write a query to display the number of books in the FACT system (Figure P7.78).

FIGURE P7.78 NUMBER OF BOOKS

Number of Books
20

79. Write a query to display the number of different book subjects in the FACT system (Figure P7.79).

FIGURE P7.79 NUMBER OF DIFFERENT SUBJECTS

Number of Subjects
4

80. Write a query to display the number of books that are available (not currently checked out) (Figure P7.80).

FIGURE P7.80 NUMBER OF BOOKS NOT CURRENTLY CHECKED OUT

Available Books
14

81. Write a query to display the highest book cost in the system (Figure P7.81).

FIGURE P7.81 MOST EXPENSIVE BOOK PRICE

Most Expensive
129.95

82. Write a query to display the lowest book cost in the system (Figure P7.82).

FIGURE P7.82 LEAST EXPENSIVE BOOK PRICE

Least Expensive
49.95

83. Write a query to display the number of different patrons who have ever checked out a book (Figure P7.83).

FIGURE P7.83 DIFFERENT PARTONS TO CHECKOUT A BOOK

DIFFERENT PATRONS
33

84. Write a query to display the subject and the number of books in each subject. Sort the results by the number of books in descending order and then by subject name in ascending order (Figure P7.84).

FIGURE P7.84 NUMBER OF BOOKS PER SUBJECT

BOOK_SUBJECT	Books In Subject
Programming	9
Cloud	4
Database	4
Middleware	3

85. Write a query to display the author ID and the number of books written by that author. Sort the results in descending order by number of books, then in ascending order by author ID (Figure P7.85).

FIGURE P7.85 NUMBER OF BOOKS PER AUTHOR

AU_ID	Books Written
262	3
460	3
185	2
229	2
251	2
383	2
394	2
559	2
218	1
273	1
284	1
438	1
581	1
592	1
603	1

86. Write a query to display the total value of all books in the library (Figure P7.86).

FIGURE P7.86 TOTAL OF ALL BOOKS

Library Value
1499

87. Write a query to display the patron ID, book number, and days kept for each check-out. "Days Kept" is the difference from the date on which the book is returned to the date it was checked out. Sort the results by days kept in descending order, then by patron ID, and then by book number (Figure P7.87). (68 rows)

FIGURE P7.87 DAYS KEPT

PATRON	BOOK	Days Kept
1165	5235	9
1209	5238	5
1160	5240	9
1160	5237	3
1202	5236	8
1203	5235	8
1174	5244	3
1181	5248	1
1170	5242	4
1161	5235	0

88. Write a query to display the patron ID, patron full name, and patron type for each patron sorted by patron ID (Figure P7.88). (50 rows)

FIGURE P7.88 PATRON AND PATRON TYPE

PAT_ID	Patron Name	PAT_TYPE
1160	robert carter	Faculty
1161	Kelsey Koch	Faculty
1165	Cedric Baldwin	Faculty
1166	Vera Alvarado	Student
1167	Alan Martin	FACULTY
1170	Cory Barry	faculty
1171	Peggy Marsh	STUDENT

89. Write a query to display the book number, title with year, and subject for each book. Sort the results by the book number (Figure P7.89). (20 rows)

FIGURE P7.89 BOOK TITLE WITH YEAR

BOOK_NUM	BOOK	BOOK_SUBJECT
5235	Beginner's Guide to JAVA (2014)	Programming
5236	Database in the Cloud (2014)	Cloud
5237	Mastering the database environment (2015)	Database
5238	Conceptual Programming (2015)	Programming
5239	J++ in Mobile Apps (2015)	Programming
5240	iOS Programming (2015)	Programming
5241	JAVA First Steps (2015)	Programming
5242	C# in Middleware Deployment (2015)	Middleware
5243	DATABASES in Theory (2015)	Database
5244	Cloud-based Mobile Applications (2015)	Cloud

90. Write a query to display the author last name, author first name, and book number for each book written by that author. Sort the results by author last name, first name, and then book number (Figure P7.90). (25 rows)

FIGURE P7.90 BOOKS WRITTEN BY AUTHOR

AU_LNAME	AU_FNAME	BOOK_NUM
Reeves	Benson	5237
Reeves	Benson	5253
Beatney	Rachel	5240
Salvadore	Carmine	5239
Salvadore	Carmine	5248
Bruer	Hugo	5243
Bruer	Hugo	5246
Chiang	Xia	5244
Chiang	Xia	5249
Chiang	Xia	5252
Durante	Reba	5235
Tankersly	Trina	5244

91. Write a query to display the author ID, book number, title, and subject for each book. Sort the results by book number and then author ID (Figure P7.91). (25 rows)

FIGURE P7.91 AUTHORS OF BOOKS

AU_ID	BOOK_NUM	BOOK_TITLE	BOOK_SUBJECT
273	5235	Beginner's Guide to JAVA	Programming
383	5236	Database in the Cloud	Cloud
185	5237	Mastering the database environment	Database
603	5238	Conceptual Programming	Programming
229	5239	J++ in Mobile Apps	Programming
460	5239	J++ in Mobile Apps	Programming
592	5239	J++ in Mobile Apps	Programming
218	5240	iOS Programming	Programming
460	5241	JAVA First Steps	Programming
559	5241	JAVA First Steps	Programming

92. Write a query to display the author last name, first name, book title, and replacement cost for each book. Sort the results by book number and then author ID (Figure P7.92). (25 rows)

FIGURE P7.92 AUTHOR NAME AND BOOK TITLE

AU_LNAME	AU_FNAME	BOOK_TITLE	BOOK_COST
Durante	Reba	Beginner's Guide to JAVA	59.95
Walsh	Neal	Database in the Cloud	79.95
Reeves	Benson	Mastering the database environment	89.95
Palca	Julia	Conceptual Programming	59.95
Salvadore	Carmine	J++ in Mobile Apps	49.95
Paulsen	Connie	J++ in Mobile Apps	49.95
Sheel	Lawrence	J++ in Mobile Apps	49.95
Beatney	Rachel	iOS Programming	79.95
Paulsen	Connie	JAVA First Steps	49.95
McGill	Rachel	JAVA First Steps	49.95
Aggerwal	Manish	C# in Middleware Deployment	59.95

93. Write a query to display the patron ID, book number, patron first name and last name, and book title for all currently checked out books. (Remember to use the redundant relationship described in the assignment instructions for current check-outs.) Sort the output by patron last name and book title (Figure P7.93).

FIGURE P7.93 CURRENTLY CHECKED OUT BOOKS

PAT_ID	BOOK_NUM	PAT_FNAME	PAT_LNAME	BOOK_TITLE
1229	5252	Gerald	Burke	Beyond the Database Veil
1229	5238	Gerald	Burke	Conceptual Programming
1228	5242	Horner	Goodman	C# in Middleware Deployment
1212	5240	Iva	McClain	iOS Programming
1172	5246	Tony	Miles	Capture the Cloud
1207	5249	Iva	Ramos	Starlight Applications

94. Write a query to display the patron ID, full name (first and last), and patron type for all patrons. Sort the results by patron type and then by last name and first name. Ensure that all sorting is case insensitive (Figure P7.94). (50 rows)

FIGURE P7.94 SORTED PATRONS WITH FULL NAMES

PAT_ID	NAME	PAT_TYPE
1165	Cedric Baldwin	Faculty
1170	Cory Barry	faculty
1160	robert carter	Faculty
1183	Helena Hughes	Faculty
1161	Kelsey Koch	Faculty
1167	Alan Martin	FACULTY
1166	Vera Alvarado	Student
1202	Holly Anthony	Student
1180	Nadine Blair	STUDENT

95. Write a query to display the book number and the number of times each book has been checked out. Do not include books that have never been checked out. Sort the results by the number of times checked out in descending order and then by book number in descending order (Figure P7.95).

FIGURE P7.95 TIMES CHECKED OUT

BOOK_NUM	Times Checked Out
5236	12
5235	9
5240	7
5238	6
5237	5
5254	4
5252	4
5249	4
5246	4
5244	4
5242	4
5248	3
5243	2

96. Write a query to display the author ID, first and last name, book number, and book title of all books in the subject "Cloud." Sort the results by book title and then by author last name (Figure P7.96).

FIGURE P7.96 BOOKS ON CLOUD COMPUTING

AU_ID	AU_FNAME	AU_LNAME	BOOK_NUM	BOOK_TITLE
251	Hugo	Bruer	5246	Capture the Cloud
252	Xia	Chiang	5244	Cloud-based Mobile Applications
284	Trina	Tankersly	5244	Cloud-based Mobile Applications
383	Neal	Walsh	5236	Database in the Cloud
252	Xia	Chiang	5249	Starlight Applications

97. Write a query to display the book number, title, author last name, author first name, patron ID, last name, and patron type for all books currently checked out to a patron. Sort the results by book title (Figure P7.97).

FIGURE P7.97 CURRENTLY CHECKED OUT BOOKS WITH AUTHORS

BOOK_NUM	BOOK_TITLE	AU_LNAME	AU_FNAME	PAT_ID	PAT_LNAME	PAT_TYPE
5252	Beyond the Database Veil	Chiang	Xia	1229	Burke	Student
5242	C# in Middleware Deployment	Aggerwal	Manish	1228	Goodman	Student
5246	Capture the Cloud	Bruer	Hugo	1172	Miles	STUDENT
5238	Conceptual Programming	Palca	Julia	1229	Burke	Student
5240	iOS Programming	Beatney	Rachel	1212	McClain	Student
5249	Starlight Applications	Chiang	Xia	1207	Ramos	Student

98. Write a query to display the book number, title, and number of times each book has been checked out. Include books that have never been checked out. Sort the results in descending order by the number of times checked out and then by title (Figure P7.98).

FIGURE P7.98 NUMBER OF CHECKOUTS FOR EVERY BOOK

BOOK_NUM	BOOK_TITLE	Times Checked Out
5236	Database in the Cloud	12
5235	Beginner's Guide to JAVA	9
5240	iOS Programming	7
5238	Conceptual Programming	6
5237	Mastering the database environment	5
5252	Beyond the Database Veil	4
5242	C# in Middleware Deployment	4
5246	Capture the Cloud	4
5244	Cloud-based Mobile Applications	4
5254	Coding Style for Maintenance	4
5249	Starlight Applications	4
5248	What You Always Wanted to Know About Database, But Were Afraid to Ask	3
5243	DATABASES in Theory	2
5239	J++ in Mobile Apps	0
5241	JAVA First Steps	0
5250	Reengineering the Middle Tier	0
5247	Shining Through the Cloud: Sun Programming	0
5245	The Golden Road to Platform independence	0
5251	Thoughts on Revitalizing Ruby	0
5253	Virtual Programming for Virtual Environments	0

99. Write a query to display the book number, title, and number of times each book has been checked out. Limit the results to books that have been checked out more than five times. Sort the results in descending order by the number of times checked out and then by title (Figure P7.99).

FIGURE P7.99 BOOKS WITH MORE THAN FIVE CHECKOUTS

BOOK_NUM	BOOK_TITLE	Times Checked Out
5236	Database in the Cloud	12
5235	Beginner's Guide to JAVA	9
5240	iOS Programming	7
5238	Conceptual Programming	6

100. Write a query to display the author ID, author last name, book title, checkout date, and patron last name for all the books written by authors with the last name "Bruer" that have ever been checked out by patrons with the last name "Miles." Sort the results by check out date (Figure P7.100).

FIGURE P7.100 BOOKS BY AUTHOR FOR PATRON "MILES"

AU_ID	AU_LNAME	BOOK_TITLE	CHECK_OUT_DATE	PAT_LNAME
251	Bruer	Capture the Cloud	4/21/2017	Miles
251	Bruer	Capture the Cloud	5/15/2017	Miles

101. Write a query to display the patron ID, first and last name of all patrons who have never checked out any book. Sort the result by patron last name and then first name (Figure P7.101).

FIGURE P7.101 PATRONS WHO NEVER CHECKED OUT A BOOK

PAT_ID	PAT_FNAME	PAT_LNAME
1166	Vera	Alvarado
1180	Nadine	Blair
1238	Erika	Bowen
1208	Ollie	Cantrell
1227	Alicia	Dickson
1205	Claire	Gomez
1239	Elton	Irwin
1240	Jan	Joyce
1243	Roberto	Kennedy
1242	Mario	King
1237	Brandi	Larson
1167	Alan	Martin
1182	Jamal	Melendez
1201	Shelby	Noble
1244	Leon	Richmond
1200	Lorenzo	Torres
1241	Irene	West

102. Write a query to display the patron ID, last name, number of times that patron has ever checked out a book, and the number of different books the patron has ever checked out. For example, if a given patron has checked out the same book twice, that would count as two checkouts but only one book. Limit the results to only patrons who have made at least three checkouts. Sort the results in descending order by number of books, then in descending order by number of checkouts, and then in ascending order by patron ID (Figure P7.102).

FIGURE P7.102 CHECKOUTS AND BOOKS BY PATRON

PAT_ID	PAT_LNAME	NUM CHECKOUTS	NUM DIFFERENT BOOKS
1161	Koch	3	3
1165	Baldwin	3	3
1181	Horne	3	3
1185	Yang	3	3
1210	Cooley	3	3
1229	Burke	3	3
1160	carter	3	2
1171	Marsh	3	2
1172	Miles	3	2
1207	Ramos	3	2
1209	Mathis	3	2
1183	Hughes	3	1

103. Write a query to display the average number of days a book is kept during a checkout (Figure P7.103).

FIGURE P7.103 AVERAGE DAYS KEPT

Average Days Kept
4.44

104. Write a query to display the patron ID and the average number of days that patron keeps books during a checkout. Limit the results to only patrons who have at least three checkouts. Sort the results in descending order by the average days the book is kept (Figure P7.104).

FIGURE P7.104 AVERAGE DAYS KEPT BY PATRON

PAT_ID	Average Days Kept
1160	7
1185	6.67
1165	5.67
1207	5.5
1209	5.33
1172	4.5
1183	4.33
1181	3.67
1171	3.67
1161	3.33
1210	2.33
1229	2

105. Write a query to display the book number, title, and cost of books that have the lowest cost of any books in the system. Sort the results by book number (Figure P7.105).

FIGURE P7.105 LEAST EXPENSIVE BOOKS

BOOK_NUM	BOOK_TITLE	BOOK_COST
5239	J++ in Mobile Apps	49.95
5241	JAVA First Steps	49.95
5248	What You Always Wanted to Know About Database, But Were Afraid to Ask	49.95
5254	Coding Style for Maintenance	49.95

106. Write a query to display the author ID, first and last name for all authors who have never written a book with the subject Programming. Sort the results by author last name (Figure P7.106).

FIGURE P7.106 AUTHORS WHO HAVE NEVER WRITTEN ON PROGRAMMING

AU_ID	AU_FNAME	AU_LNAME
581	Manish	Aggerwal
251	Hugo	Bruer
262	Xia	Chiang
438	Perry	Pearson
284	Trina	Tankersly
383	Neal	Walsh

107. Write a query to display the book number, title, subject, average cost of books within that subject, and the difference between each book's cost and the average cost of books in that subject. Sort the results by book title (Figure P7.107).

FIGURE P7.107 BOOKS WITH AVERAGE COST BY SUBJECT

BOOK_NUM	BOOK_TITLE	BOOK_SUBJECT	AVGCOST	DIFFERENCE
5235	Beginner's Guide to JAVA	Programming	66.62	-6.67
5236	Database in the Cloud	Cloud	72.45	7.5
5237	Mastering the database environment	Database	84.95	5
5238	Conceptual Programming	Programming	66.62	-6.67
5239	J++ in Mobile Apps	Programming	66.62	-16.67
5240	iOS Programming	Programming	66.62	13.33
5241	JAVA First Steps	Programming	66.62	-16.67
5242	C# in Middleware Deployment	Middleware	89.95	-30
5243	DATABASES in Theory	Database	84.95	45
5244	Cloud-based Mobile Applications	Cloud	72.45	-2.5
5245	The Golden Road to Platform independence	Middleware	89.95	30
5246	Capture the Cloud	Cloud	72.45	-2.5
5247	Shining Through the Cloud: Sun Programming	Programming	66.62	43.33
5248	What You Always Wanted to Know About Database, But Were Afraid to Ask	Database	84.95	-35
5249	Starlight Applications	Cloud	72.45	-2.5
5250	Reengineering the Middle Tier	Middleware	89.95	0
5251	Thoughts on Revitalizing Ruby	Programming	66.62	-6.67
5252	Beyond the Database Veil	Database	84.95	-15
5253	Virtual Programming for Virtual Environments	Programming	66.62	13.33
5254	Coding Style for Maintenance	Programming	66.62	-16.67

108. Write a query to display the book number, title, subject, author last name, and the number of books written by that author. Limit the results to books in the Cloud subject. Sort the results by book title and then author last name (Figure P7.108).

FIGURE P7.108 NUMBER OF BOOKS BY CLOUD AUTHORS

BOOK_NUM	BOOK_TITLE	BOOK_SUBJECT	AU_LNAME	Num Books by Author
5246	Capture the Cloud	Cloud	Bruer	2
5244	Cloud-based Mobile Applications	Cloud	Chiang	3
5244	Cloud-based Mobile Applications	Cloud	Tankersly	1
5236	Database in the Cloud	Cloud	Walsh	2
5249	Starlight Applications	Cloud	Chiang	3

109. Write a query to display the lowest average cost of books within a subject and the highest average cost of books within a subject (Figure P7.109).

FIGURE P7.109 LOWEST AND HIGHEST AVERAGE SUBJECT COSTS

Lowest Avg Cost	Highest Avg Cost
66.62	89.95