Introduction to Structured Query Language

2

Chapter Objectives

- To understand the use of extracted data sets in business intelligence (BI) systems
- To understand the use of ad-hoc queries in business intelligence (BI) systems
- To understand the history and significance of Structured Query Language (SQL)
- To understand the SQL SELECT/FROM/WHERE framework as the basis for database queries
- To create SQL queries to retrieve data from a single table
- To create SQL queries that use the SQL SELECT, FROM, WHERE, ORDER BY, GROUP BY, and HAVING clauses
- To create SQL queries that use the SQL DISTINCT, AND, OR, NOT, BETWEEN, LIKE, and IN keywords
- To create SQL queries that use the SQL built-in functions of SUM, COUNT, MIN, MAX, and AVG with and without the SQL GROUP BY clause
- To create SQL queries that retrieve data from a single table while restricting the data based upon data in another table (subquery)
- To create SQL queries that retrieve data from multiple tables using the SQL join and JOIN ON operations
- To create SQL queries that retrieve data from multiple tables using the SQL OUTER JOIN operation

In today's business environment, users typically use data stored in databases to produce information that can help them make business decisions. In Chapter 12, we will take an in-depth look at **business intelligence (BI) systems,** which are information systems used to support management decisions by producing information for assessment, analysis, planning, and control. In this chapter, we will see how BI systems users use **ad-hoc queries,** which are essentially questions that can be answered using database data. For example, in English an ad-hoc query would be "How many customers in Portland, Oregon, bought our green baseball cap?" These queries are called *ad-hoc* because they are created by the user as needed, rather than programmed into an application.

This approach to database querying has become important enough that some companies produce dedicated applications to help users who are not familiar with database structures create ad-hoc queries. One example is OpenText's OpenText Business Intelligence product (*http://www.opentext.com/2/global/products/products-content-reporting/products-opentext-business-intelligence.htm*), which uses a user-friendly **graphical user interface (GUI)** to simplify the creation of ad-hoc queries. Personal databases such as Microsoft Access also have ad-hoc query tools available. Microsoft Access uses a GUI style called **query by example (QBE)** to simplify ad-hoc queries.

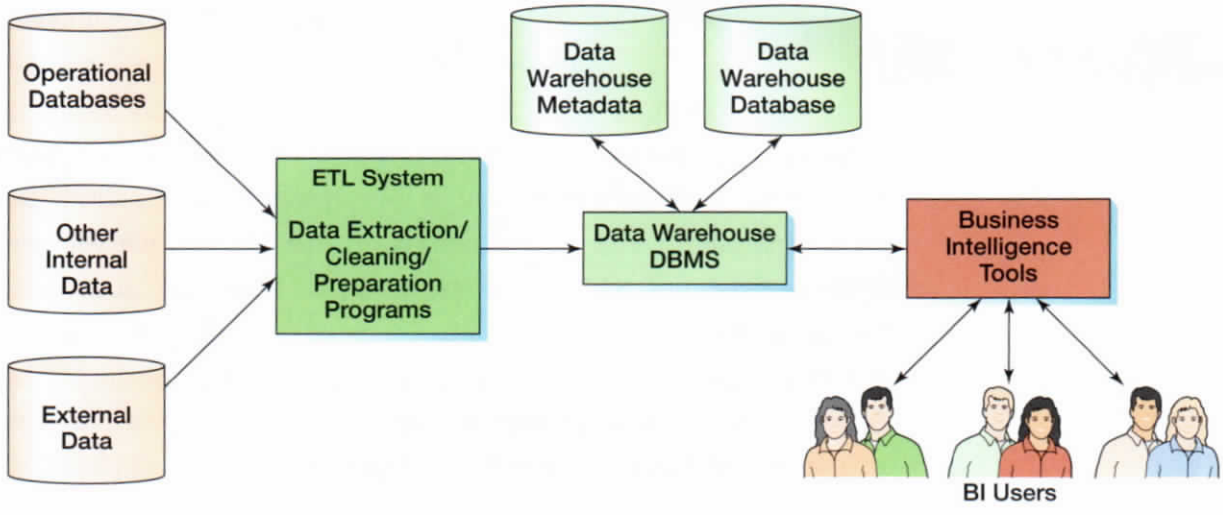
However, **Structured Query Language (SQL)**—the universal query language of relational DBMS products—is always behind the user-friendly GUIs. In this chapter, we will introduce SQL by learning how to write and run SQL queries. We will then return to SQL in Chapter 7 to learn how to use it for other purposes, such as how to create and add data to the databases themselves.

Components of a Data Warehouse

BI systems typically store their associated data in **data warehouses**, which are database systems that have data, programs, and personnel that specialize in the preparation of data for BI processing. Data warehouses will be discussed in detail in Chapter 12, and for now we will simply note that data warehouses vary in scale and scope. They can be as simple as a sole employee processing a data extract on a part-time basis or as complex as a department with dozens of employees maintaining libraries of data and programs.

Figure 2-1 shows the components of a typical company-wide data warehouse. Data are read from operational databases (the databases that store the company's current day-to-day transaction data), from other internal data, or from external data source by the **Extract, Transform, and Load (ETL) system**. The ETL system then cleans and prepares the data for BI processing. This can be a complex process, but the data is then stored in the **data**

Figure 2-1
Components of a Data Warehouse



warehouse DBMS for use by BI users who access the data by various BI tools. As described in Chapter 1, the DBMS used for the data warehouse stores both databases and the metadata for those databases.

BY THE WAY

A small, specialized data warehouse is referred to as a **data mart.** Data marts and their relationship to data warehouses are discussed in Chapter 12. Note that the DBMS used for the data warehouse may or may not be the same DBMS product used for the operational databases. For example, operational databases may be stored in an Oracle Database 11*g* Release 2 DBMS, while the data warehouse uses a Microsoft SQL Server 2012 DBMS.

Cape Codd Outdoor Sports

For our work in this chapter, we will use data from Cape Codd Outdoor Sports (although based on a real outdoor retail equipment vendor, Cape Codd Outdoor Sports is a fictitious company). Cape Codd sells recreational outdoor equipment in 15 retail stores across the United States and Canada. It also sells merchandise over the Internet from a Web storefront application and via mail order. All retail sales are recorded in a sales database managed by the Oracle Database 11*g* Release 2 DBMS, as shown in Figure 2-2.

The Extracted Retail Sales Data

Cape Codd's marketing department wants to perform an analysis of in-store sales. Accordingly, marketing analysts ask the IT department to extract retail sales data from the operational database. To perform the marketing study, they do not need all of the order data. They want just the tables and columns shown in Figure 2-3. Looking at this figure, it is easy to see that columns that would be needed in an operational sales database are *not* included in the extracted data. For example, the RETAIL_ORDER table does *not* have

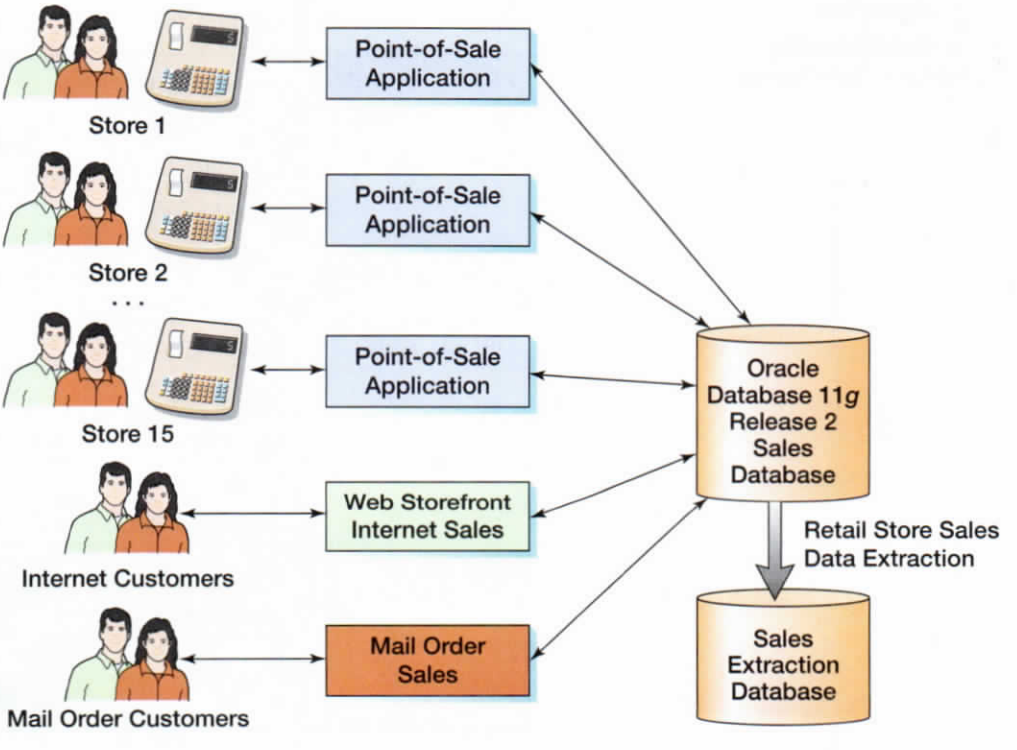


Figure 2-2
The Cape Codd Retail Sales Data Extraction Process

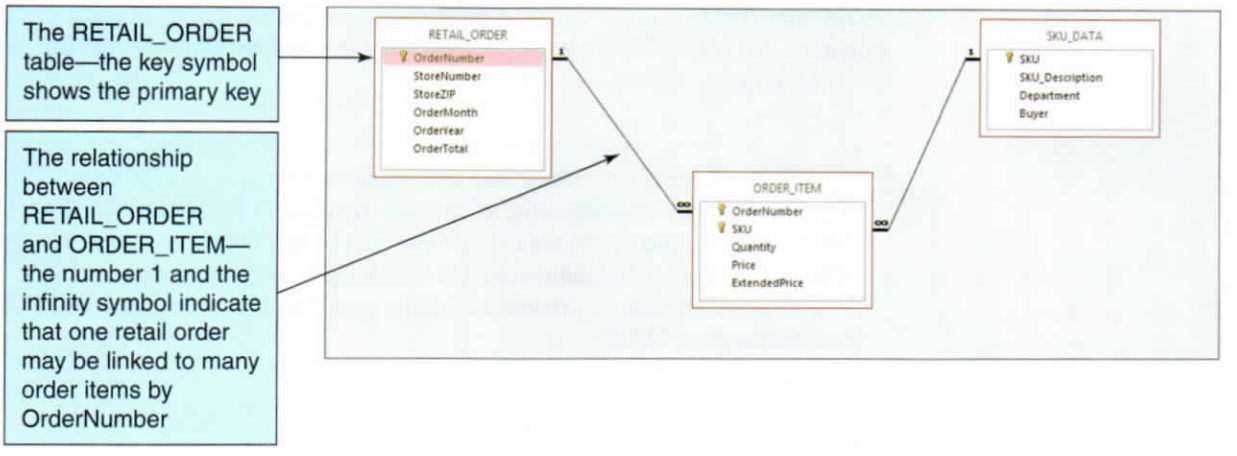


Figure 2-3
Cape Codd Extracted Retail Sales Data Database Tables and Relationships

CustomerLastName, CustomerFirstName, and OrderDay columns. The data types for the columns in the tables is shown in Figure 2-4.

As shown in Figures 2-3 and 2-4, three tables are needed: RETAIL_ORDER, ORDER_ITEM, and SKU_DATA. The RETAIL_ORDER table has data about each retail sales order, the ORDER_ITEM table has data about each item in an order, and the SKU_DATA table has data about each **stock-keeping unit (SKU)**. SKU is a unique identifier for each particular item that Cape Codd sells. The data stored in the tables is shown in Figure 2-5.

BY THE WAY

The dataset shown is a small dataset we are using to illustrate the concepts explained in this chapter. A "real world" data extract would produce a much larger dataset.

RETAIL_ORDER Data

As shown in Figures 2-3, 2-4, and 2-5, the RETAIL_ORDER table has columns for OrderNumber, StoreNumber, StoreZIP (the ZIP code of the store selling the order), OrderMonth, OrderYear,

Figure 2-4
Cape Codd Extracted Retail Sales Data Format

Table	Column	Date Type
RETAIL_ORDER	OrderNumber	Integer
	StoreNumber	Integer
	StoreZIP	Character (9)
	OrderMonth	Character (12)
	OrderYear	Integer
	OrderTotal	Currency
ORDER_ITEM	OrderNumber	Integer
	SKU	Integer
	Quantity	Integer
	Price	Currency
	ExtendedPrice	Currency
SKU_DATA	SKU	Integer
	SKU_Description	Character (35)
	Department	Character (30)
	Buyer	Character (30)

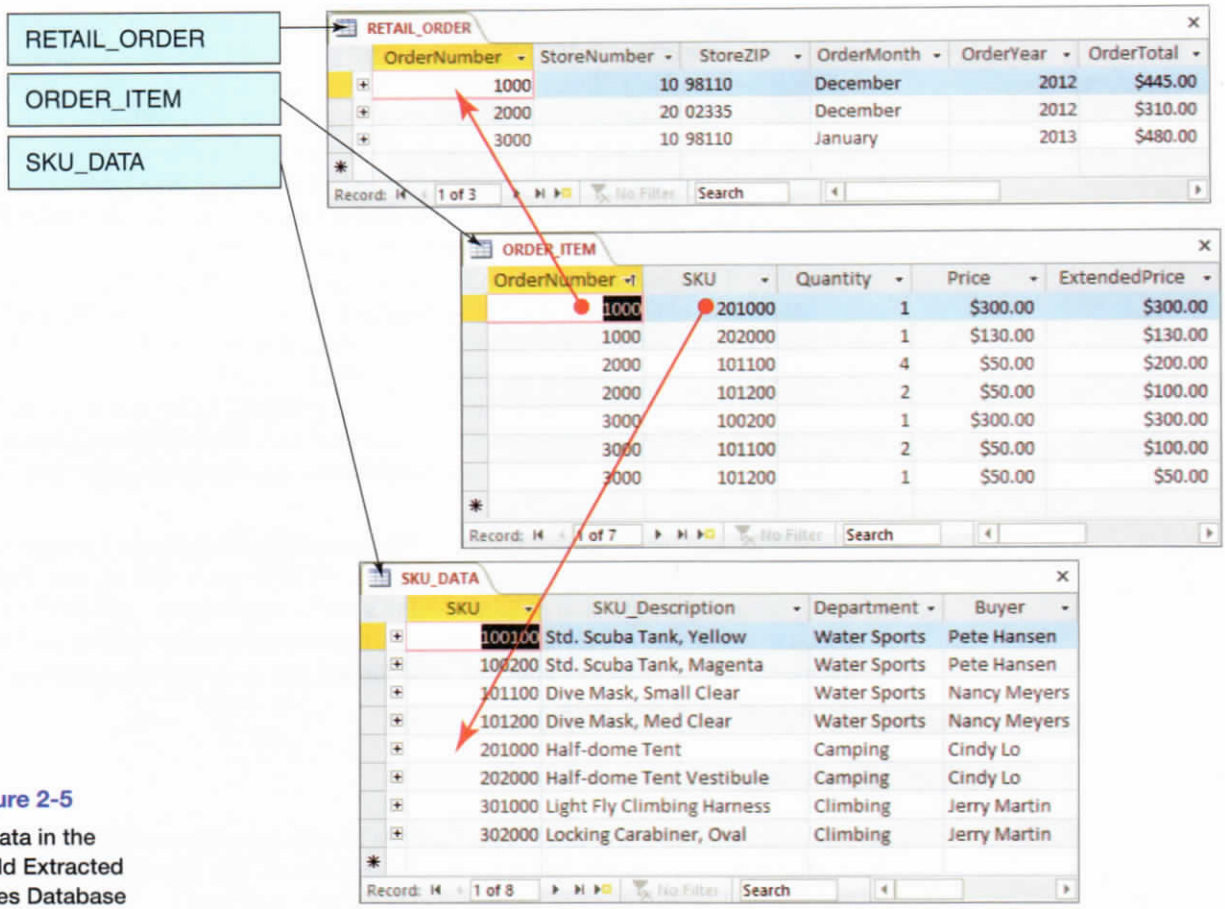


Figure 2-5
Sample Data in the Cape Codd Extracted Retail Sales Database

and OrderTotal. We can write this information in the following format, with OrderNumber underlined to show that it is the primary key of the RETAIL_ORDER table:

RETAIL_ORDER (OrderNumber, StoreNumber, StoreZIP, OrderMonth, OrderYear, OrderTotal)

Sample data for RETAIL_ORDER are shown in Figure 2-5. This extract only includes data for retail store sales, and operational data for other types of sales (and returns and other sales-related transactions) are not copied during the extraction process. Further, the data extraction process selects only a few columns of the operational data—the Point of Sale (POS) and other sales applications process far more data than that shown here. The operational database also stores the data in a different format. For example, the order data in the Oracle Database 11*g* Release 2 operational database contain a column named OrderDate that stores the data in the date format MM/DD/YYYY (e.g., 10/22/2010 for October 22, 2010). The extraction program used to populate the retail sales extracted data database converts OrderDate into two separate values of OrderMonth and OrderYear. This is done because this is the data format that marketing wants. Such filtering and data transformation are typical of a data extraction process.

ORDER_ITEM Data

As shown in Figures 2-3, 2-4, and 2-5, the ORDER_ITEM table has columns for OrderNumber, SKU, Quantity, Price, and ExtendedPrice (which equals Quantity × Price). We can write this information in the following format, with both OrderNumber and SKU underlined to show that together they are the composite primary key of the ORDER_ITEM table, and with them also italicized to show that they are also foreign keys:

ORDER_ITEM (*OrderNumber*, *SKU*, Quantity, Price, ExtendedPrice)

Thus, the ORDER_ITEM table stores an extract of the items purchased in each order. There is one row in the table for each item in an order, and this item is identified by its SKU. To understand this table, think about a sales receipt you get from a retail store. That receipt has data for one order. It

includes basic order data such as the date and order total, and it has one line for each item you purchase. The rows in the ORDER_ITEM table correspond to the lines on such an order receipt.

The OrderNumber Column in ORDER_ITEM relates each row in ORDER_ITEM to the corresponding OrderNumber in the RETAIL_ORDER table. SKU identifies the actual item purchased by its stock-keeping unit number. Further, the SKU column in ORDER_ITEM relates each row in ORDER_ITEM to its corresponding SKU in the SKU_DATA table (discussed in the next section). Quantity is the number of items of that SKU purchased in that order. Price is the price of each item, and ExtendedPrice is equal to Quantity × Price.

ORDER_ITEM data are shown in the bottom part of Figure 2-5. The first row relates to order 1000 and to SKU 201000. For SKU 201000, one item was purchased for $300.00, and the ExtendedPrice was $300.00. The second row shows the second item in order 1000. There, 1 of item 202000 was purchased for $50.00, and the ExtendedPrice is 1 × $50.00, or $50.00. This table structure of an ORDER table related to an ORDER_ITEM table is typical for a sales system with many items in one order. We will discuss it in detail in Chapters 5 and 6, where we will create a data model of a complete order and then design the database for that data model.

BY THE WAY

You would expect the total of ExtendedPrice for all rows for a given order to equal OrderTotal in the RETAIL_ORDER table. It does not. For order 1000, for example, the sum of ExtendedPrice in the relevant rows of ORDER_ITEM is $300.00 + $130.00 = $430.00. However, the OrderTotal for order 1000 is $445.00. The difference occurs because OrderTotal includes tax, shipping, and other charges that do not appear in the data extract.

SKU_DATA Table

As shown in Figures 2-3, 2-4, and 2-5, the SKU_DATA table has columns SKU, SKU_Description, Department, and Buyer. We can write this information in the following format, with SKU underlined to show that it is the primary key of the SKU_DATA table:

SKU_DATA (SKU, SKU_Description, Department, Buyer)

SKU is an integer value that identifies a particular product sold by Cape Codd. For example, SKU 100100 identifies a yellow, standard-size SCUBA tank, whereas SKU 100200 identifies the magenta version of the same tank. SKU_Description contains a brief text description of each item. Department and Buyer identify the department and individual who is responsible for purchasing the product. As with the other tables, these columns are a subset of the SKU data stored in the operational database.

The Complete Cape Codd Data Extract Schema

A database **schema** is a complete logical view of the database, containing all the tables, all the columns in each table, the primary key of each table (indicated by underlining the column names of the primary key columns), and the foreign keys that link the tables together (indicated by italicizing the column names of the foreign key columns). The schema for the Cape Codd sales data extract therefore is:

RETAIL_ORDER (OrderNumber, StoreNumber, StoreZIP, OrderMonth, OrderYear, OrderTotal)

ORDER_ITEM (*OrderNumber*, *SKU*, Quantity, Price, ExtendedPrice)

SKU_DATA (SKU, SKU_Description, Department, Buyer)

Note how the composite primary key for ORDER_ITEM also contains the foreign keys linking this table to RETAIL_ORDER and SKU_DATA.

BY THE WAY

In the Review Questions at the end of this chapter, we will extend this schema to include two additional tables: WAREHOUSE and INVENTORY. The figures in this chapter include these two tables in the Cape Codd database, but they are not used in our discussion of SQL in the chapter text.

Data Extracts Are Common

Before we continue, realize that the data extraction process described here is not just an academic exercise. To the contrary, such extraction processes are realistic, common, and important BI system operations. Right now, hundreds of businesses worldwide are using their BI systems to create extract databases just like the one created by Cape Codd.

In the next sections of this chapter, you will learn how to write SQL statements to process the extracted data via ad hoc **SQL queries**, which is how SQL is used to "ask questions" about the data in the database. This knowledge is exceedingly valuable and practical. Again, right now, as you read this paragraph, hundreds of people are writing SQL to create information from extracted data. The SQL you will learn in this chapter will be an essential asset to you as a knowledge worker, application programmer, or database administrator. Invest the time to learn SQL—the investment will pay great dividends later in your career.

SQL Background

SQL was developed by the IBM Corporation in the late 1970s. It was endorsed as a national standard by the **American National Standards Institute (ANSI)** in 1986 and by the **International Organization for Standardization (ISO)** (and no, that's not a typo—the acronym is *ISO*, not *IOS*!) in 1987. Subsequent versions of SQL were adopted in 1989 and 1992. The 1992 version is sometimes referred to as SQL-92 and sometimes as ANSI-92 SQL. In 1999, SQL:1999 (also referred to as SQL3), which incorporated some object-oriented concepts, was released. This was followed by the release of SQL:2003 in 2003; SQL:2006 in 2006; SQL:2008 in 2008; and, most recently, SQL:2011 in 2011. Each of these added new features or extended existing SQL features, the most important of which for us are the SQL standardization of the INSTEAD OF trigger (SQL triggers are discussed in Chapter 7) in SQL:2008 and the support for **Extensible Markup Language (XML)** (XML is discussed in Chapter 11) added in SQL:2009. Our discussions in this chapter and in Chapter 7 mostly focus on common language features that have been in SQL since SQL-92, but does include some features from SQL:2003 and SQL:2008. We discuss the SQL XML features in Chapter 11.

BY THE WAY

Although there is an SQL *standard*, that does not mean the that SQL is *standardized* across DBMS products! Indeed, each DBMS implements SQL in its own peculiar way, and you will have to learn the idiosyncrasies of the SQL dialect your DBMS uses.

In this book, we are using Microsoft's SQL Server 2012 SQL syntax, with some limited discussion of the different SQL dialects. The Oracle Database 11*g* Release 2 SQL syntax is used in Chapter 10B, and the MySQL SQL 5.6 SQL syntax is used in Chapter 10C.

SQL is not a complete programming language, like Java or C#. Instead, it is called a **data sublanguage** because it has only those statements needed for creating and processing database data and metadata. You can use SQL statements in many different ways. You can submit them directly to the DBMS for processing. You can embed SQL statements into client/server application programs. You can embed them into Web pages, and you can use them in reporting and data extraction programs. You also can execute SQL statements directly from Visual Studio.NET and other development tools.

SQL statements are commonly divided into categories, five of which are of interest to us here:

- **Data definition language (DDL)** statements, which are used for creating tables, relationships, and other structures
- **Data manipulation language (DML)** statements, which are used for querying, inserting, modifying, and deleting data
- **SQL/Persistent stored modules (SQL/PSM)** statements, which extend SQL by adding procedural programming capabilities, such as variables and flow-of-control statements, that provide some programmability within the SQL framework.

- **Transaction control language (TCL)** statements, which are used to mark transaction boundaries and control transaction behavior.
- **Data control language (DCL)** statements, which are used to grant database permissions (or to revoke those permissions) to users and groups, so that the users or groups can perform various operations on the data in the database

This chapter considers only DML statements for querying data. The remaining DML statements for inserting, modifying, and deleting data are discussed in Chapter 7, where we will also discuss SQL DDL statements. SQL/PSM is introduced in Chapter 7, and the specific variations of it used with each DBMS are discussed in detail in Chapter 10A for Microsoft SQL Server 2012, Chapter 10B for Oracle Database 11*g* Release 2, and Chapter 10C for MySQL 5.6. TCL and DCL statements are discussed in Chapter 9.

BY THE WAY

Some authors treat SQL queries as a separate part of SQL rather than as a part of SQL DML. We note that the SQL/Framework section of the SQL specification includes queries as part of the "SQL-data statements" class of statements along with the rest of the SQL DML statements and treat them as SQL DML statements.

BY THE WAY

The four actions listed for SQL DML are sometimes referred to as **CRUD**: create, read, update, and delete. We do *not* use this term in this book, but now you know what it means.

SQL is ubiquitous, and SQL programming is a critical skill. Today, nearly all DBMS products process SQL, with the only exceptions being some of the emerging NoSQL and Big Data movement products. Enterprise-class DBMSs such as Microsoft SQL Server 2012, Oracle Database 11*g* Release 2, Oracle MySQL 5.6, and IBM DB2 require that you know SQL. With these products, all data manipulation is expressed using SQL.

As explained in Chapter 1, if you have used Microsoft Access, you have used SQL, even if you didn't know it. Every time you process a form, create a report, or run a query, Microsoft Access generates SQL and sends that SQL to Microsoft Access's internal ADE DBMS engine. To do more than elementary database processing, you need to uncover the SQL hidden by Microsoft Access. Further, once you know SQL, you will find it easier to write a query statement in SQL rather than fight with the graphical forms, buttons, and other paraphernalia that you must use to create queries with the Microsoft Access query-by-example style GUI.

The SQL SELECT/FROM/WHERE Framework

This section introduces the fundamental statement framework for SQL query statements. After we discuss this basic structure, you will learn how to submit SQL statements to Microsoft Access, Microsoft SQL Server, Oracle Database, and MySQL. If you choose, you can then follow along with the text and process the SQL statements as they are explained in the rest of this chapter. The basic form of SQL queries uses the **SQL SELECT/FROM/WHERE framework**. In this framework:

- The **SQL SELECT clause** specifies which *columns* are to be listed in the query results.
- The **SQL FROM clause** specifies which *tables* are to be used in the query.
- The **SQL WHERE clause** specifies which *rows* are to be listed in the query results.

Let's work through some examples so that this framework makes sense to you.

Reading Specified Columns from a Single Table

We begin very simply. Suppose we want to obtain just the values of the Department and Buyer columns of the SKU_DATA table. An SQL statement to read that data is the following:

```
SELECT    Department, Buyer
FROM      SKU_DATA;
```

Using the data in Figure 2-5, when the DBMS processes this statement the result will be:

	Department	Buyer
1	Water Sports	Pete Hansen
2	Water Sports	Pete Hansen
3	Water Sports	Nancy Meyers
4	Water Sports	Nancy Meyers
5	Camping	Cindy Lo
6	Camping	Cindy Lo
7	Climbing	Jerry Martin
8	Climbing	Jerry Martin

When SQL statements are executed, the statements transform tables. SQL statements start with a table, process that table in some way, and then place the results in another table structure. Even if the result of the processing is just a single number, that number is considered to be a table with one row and one column. As you will learn at the end of this chapter, some SQL statements process multiple tables. Regardless of the number of input tables, though, the result of every SQL statement is a single table.

Notice that SQL statements terminate with a semicolon (;) character. The semicolon is required by the SQL standard. Although some DBMS products will allow you to omit the semicolon, some will not, so develop the habit of terminating SQL statements with a semicolon.

SQL statements can also include an **SQL comment**, which is a block of text that is used to document the SQL statement while not executed as part of the SQL statement. SQL comments are enclosed in the symbols /* and */, and any text between these symbols is ignored when the SQL statement is executed. For example, here is the previous SQL query with an SQL comment added to document the query by including a query name:

```
/* *** SQL-Query-CH02-01 *** */
SELECT     Department, Buyer
FROM       SKU_DATA;
```

Because the SQL comment is ignored when the SQL statement is executed, the output from this query is identical to the query output shown above. We will use similar comments to label the SQL statements in this chapter as an easy way to reference a specific SQL statement in the text.

Specifying Column Order in SQL Queries from a Single Table

The order of the column names in the SELECT phrase determines the order of the columns in the results table. Thus, if we switch Buyer and Department in the SELECT phrase, they will be switched in the output table as well. Hence, the SQL statement:

```
/* *** SQL-Query-CH02-02 *** */
SELECT     Buyer, Department
FROM       SKU_DATA;
```

produces the following result table:

	Buyer	Department
1	Pete Hansen	Water Sports
2	Pete Hansen	Water Sports
3	Nancy Meyers	Water Sports
4	Nancy Meyers	Water Sports
5	Cindy Lo	Camping
6	Cindy Lo	Camping
7	Jerry Martin	Climbing
8	Jerry Martin	Climbing

Notice that some rows are duplicated in these results. The data in the first and second row, for example, are identical. We can eliminate duplicates by using the **SQL DISTINCT keyword**, as follows:

```
/* *** SQL-Query-CH02-03 *** */
SELECT     DISTINCT Buyer, Department
FROM       SKU_DATA;
```

The result of this statement, where all of the duplicate rows have been removed, is:

	Buyer	Department
1	Cindy Lo	Camping
2	Jerry Martin	Climbing
3	Nancy Meyers	Water Sports
4	Pete Hansen	Water Sports

The reason that SQL does not automatically eliminate duplicate rows is that it can be very time consuming to do so. To determine if any rows are duplicates, every row must be compared with every other row. If there are 100,000 rows in a table, that checking will take a long time. Hence, by default duplicates are not removed. However, it is always possible to force their removal using the DISTINCT keyword.

Suppose that we want to view all of the columns of the SKU_DATA table. To do so, we can name each column in the SELECT statement as follows:

```
/* *** SQL-Query-CH02-04 *** */
SELECT     SKU, SKU_Description, Department, Buyer
FROM       SKU_DATA;
```

The result will be a table with all of the rows and all four of the columns in SKU_DATA:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	201000	Half-dome Tent	Camping	Cindy Lo
6	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
7	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
8	302000	Locking Carabiner, Oval	Climbing	Jerry Martin

However, SQL provides a shorthand notation for querying all of the columns of a table. The shorthand is to the **SQL asterisk (*) wildcard character** to indicate that we want all the columns to be displayed:

```
/* *** SQL-Query-CH02-05 *** */
SELECT     *
FROM       SKU_DATA;
```

The result will again be a table with all rows and all four of the columns in SKU_DATA:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	201000	Half-dome Tent	Camping	Cindy Lo
6	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
7	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
8	302000	Locking Carabiner, Oval	Climbing	Jerry Martin

Reading Specified Rows from a Single Table

Suppose we want all of the *columns* of the SKU_DATA table, but we want only the *rows* for the Water Sports department. We can obtain that result by using the SQL WHERE clause as follows:

```
/* *** SQL-Query-CH02-06 *** */
SELECT      *
FROM        SKU_DATA
WHERE       Department='Water Sports';
```

The result of this statement will be:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
3	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers

In an SQL WHERE clause, if the column contains text or date data, the comparison values must be enclosed in single quotation marks ('{text or date data}'). If the column contains numeric data, however, the comparison values need not be in quotes. Thus, to find all of the SKU rows with a value greater than 200,000, we would use the SQL statement (note that no comma is included in the numeric value code):

```
/* *** SQL-Query-CH02-07 *** */
SELECT      *
FROM        SKU_DATA
WHERE       SKU > 200000;
```

The result is:

	SKU	SKU_Description	Department	Buyer
1	201000	Half-dome Tent	Camping	Cindy Lo
2	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
3	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
4	302000	Locking Carabiner, Oval	Climbing	Jerry Martin

BY THE WAY

SQL is very fussy about single quotes. It wants the plain, nondirectional quotes found in basic text editors. The fancy directional quotes produced by many word processors will produce errors. For example, the data value 'Water Sports' is correctly stated, but ‘Water Sports’ is not. Do you see the difference?

Reading Specified Columns and Rows from a Single Table

So far, we have selected certain columns and all rows and we have selected all columns and certain rows. We can combine these operations to select certain columns and certain rows by naming the columns we want and then using the SQL WHERE clause. For example, to obtain the SKU_Description and Department of all products in the Climbing department, we use the SQL query:

```
/* *** SQL-Query-CH02-08 *** */
SELECT     SKU_Description, Department
FROM       SKU_DATA
WHERE      Department='Climbing';
```

The result is:

	SKU_Description	Department
1	Light Fly Climbing Harness	Climbing
2	Locking Carabiner, Oval	Climbing

SQL does not require that the column used in the WHERE clause also appear in the SELECT clause column list. Thus, we can specify:

```
/* *** SQL-Query-CH02-09 *** */
SELECT     SKU_Description, Buyer
FROM       SKU_DATA
WHERE      Department='Climbing';
```

where the qualifying column, Department, does not appear in the SELECT clause column list. The result is:

	SKU_Description	Buyer
1	Light Fly Climbing Harness	Jerry Martin
2	Locking Carabiner, Oval	Jerry Martin

BY THE WAY

Standard practice is to write SQL statements with the SELECT, FROM, and WHERE clauses on separate lines. This practice is just a coding convention, however, and SQL parsers do not require it. You could code SQL-Query-CH02-09 all on one line as:

```
SELECT SKU_Description, Buyer FROM SKU_DATA WHERE Department=
'Climbing';
```

All DBMS products would process the statement written in this fashion. However, the standard multiline coding convention makes SQL easier to read, and we encourage you to write your SQL according to it.

BY THE WAY

When using a date in the WHERE clause, you can usually enclose it in single quotes just as you would a character string, However, when using Microsoft Access you must enclose dates with the # symbol. For example:

```
SELECT    *
FROM      PROJECT
WHERE     StartDate = #05/10/11#;
```

Oracles Database 11*g* Release 2 and MySQL 5.6 can also have idiosyncrasies when using date data in SQL statements, and this is discussed in Chapters 10B and 10C respectively.

Submitting SQL Statements to the DBMS

Before continuing the explanation of SQL, it will be useful for you to learn how to submit SQL statements to specific DBMS products. That way, you can work along with the text by keying and running SQL statements as you read the discussion. The particular means by which you submit SQL statements depends on the DBMS. Here we will describe the process for Microsoft Access 2013, Microsoft SQL Server 2012, Oracle Database 11*g* Release 2, and MySQL 5.6.

BY THE WAY

You can learn SQL without running the queries in a DBMS, so if for some reason you do not have Microsoft Access, Microsoft SQL Server, Oracle Database, or MySQL readily available, do not despair. You can learn SQL without them. Chances are your instructor, like a lot of us in practice today, learned SQL without a DBMS. It is just that SQL statements are easier to understand and remember if you can run the SQL while you read. Given that there that there are freely downloadable versions of Microsoft SQL Server 2012 Express edition, Oracle Database 11*g* Express Edition, and MySQL 5.6 Community Server, you can have an installed DBMS to run these SQL examples even if you have not purchased Microsoft Access. See Chapters 10A, 10B, and 10C for specific instructions for creating databases using each of these products. The SQL scripts needed to create the Cape Codd Outdoor Sports database used in this chapter are available at *www.pearsonhighered.com/kroenke*.

Using SQL in Microsoft Access 2013

Before you can execute SQL statements, you need a computer that has Microsoft Access installed, and you need a Microsoft Access database that contains the tables and sample data in Figure 2-5. Microsoft Access is part of many versions of the Microsoft Office suite, so it should not be too difficult to find a computer that has it.

Because Microsoft Access is commonly used in classes that use this book as a textbook, we will look at how to use SQL in Microsoft Access in some detail. Before we proceed, however, we need to discuss a specific peculiarity of Microsoft Access—the limitations of the default version of SQL used in Microsoft Access.

"Does Not Work with Microsoft Access ANSI-89 SQL"

As mentioned previously, our discussion of SQL is based on SQL features present in SQL standards since the ANSI SQL-92 standard (which Microsoft refers to as ANSI-92 SQL). Unfortunately, Microsoft Access 2013 still defaults to the earlier SQL-89 version—Microsoft calls it ANSI-89 SQL or Microsoft Jet SQL (after the Microsoft Jet DBMS engine used by Microsoft Access). ANSI-89 SQL differs significantly from SQL-92, and, therefore, some features of the SQL-92 language will not work in Microsoft Access.

Microsoft Access 2013 (and the earlier Microsoft Access 2003, 2007, and 2010 versions) does contain a setting that allows you to use SQL-92 instead of the default ANSI-89 SQL. Microsoft included this option to allow Microsoft Access tools such as forms and reports to be used in

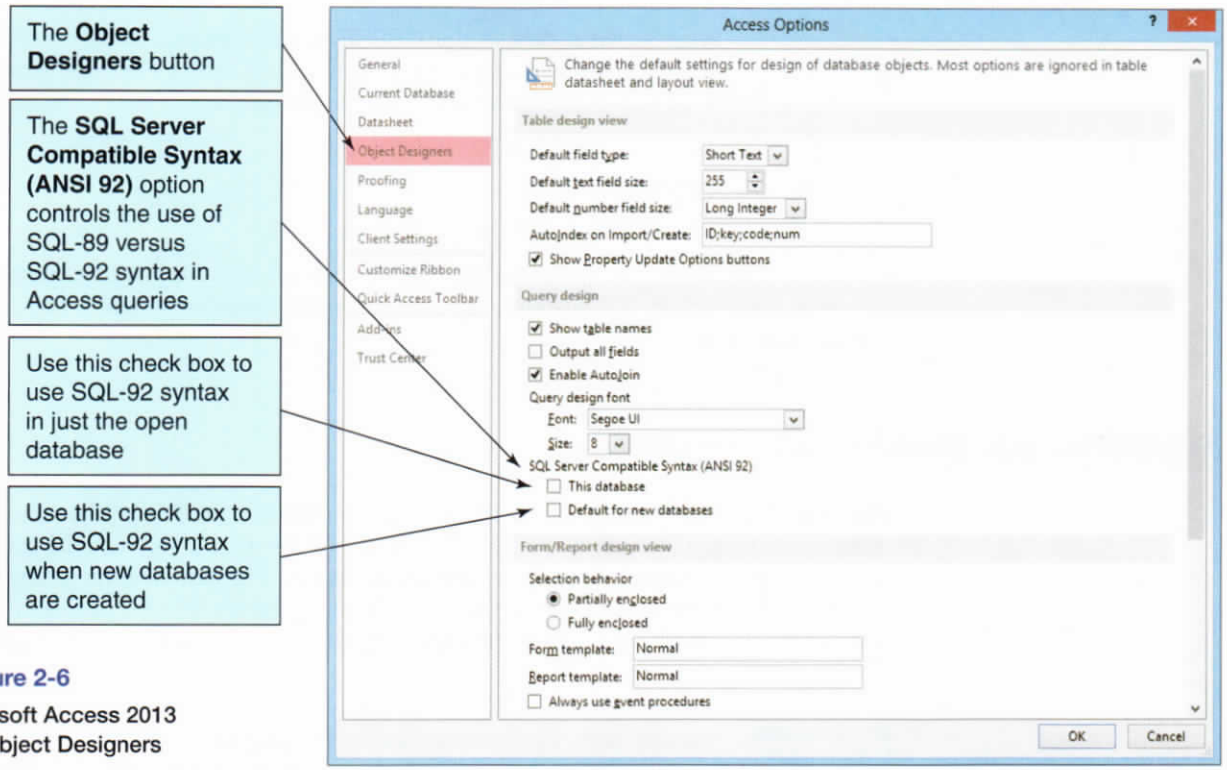


Figure 2-6
The Microsoft Access 2013 Options Object Designers Page

application development for Microsoft SQL Server, which supports newer SQL standards. To set the option in Microsoft Access 2013, click the **File** command tab and then click the **Options** command to open the Access Options dialog box. In the Access Options dialog box, click the **Object Designers** button to display the Access Options Object Designers page, as shown in Figure 2-6.

As shown in Figure 2-6, the **SQL Server Compatible Syntax (ANSI 92)** options control which version of SQL is used in a Microsoft Access 2013 database. If you check the **This database** check box, you will use SQL-92 syntax in the current database. Or you can check the **Default for new databases** check box to make SQL-92 syntax the default for all new databases you create. When you click the **OK** button to save the changed SQL syntax option, the SQL-Syntax Information dialog box shown in Figure 2-7 will be displayed. Read the information, and then click the **OK** button to close the dialog box.

Unfortunately, very few Microsoft Access users or organizations using Microsoft Access are likely to set the Microsoft Access SQL version to the SQL-92 option, and, in this chapter, we assume that Microsoft Access is running in the default ANSI-89 SQL mode. One advantage of doing so is it will help you understand the limitations of Microsoft Access ANSI-89 SQL and how to cope with them.

In the discussion that follows, we use "Does Not Work with Microsoft Access ANSI-89 SQL" boxes to identify SQL commands and SQL clauses that do not work in Microsoft Access ANSI-89 SQL. We also identify any workarounds that are available. Remember that the one *permanent* workaround is to choose to use the SQL-92 syntax option in the databases you create!

Nonetheless, two versions of the Microsoft Access 2013 Cape Codd Outdoor Sports database are available at *www.pearsonhighered.com/kroenke* for your use with this chapter. The Microsoft Access database file named *Cape-Codd.accdb* is set to use Microsoft Access ANSI-89, whereas

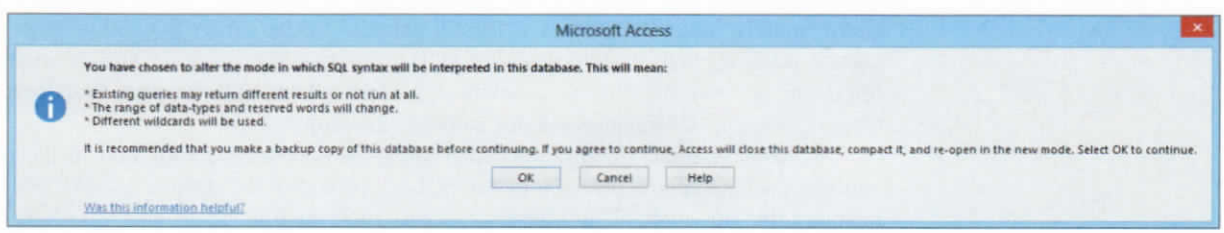


Figure 2-7
The Microsoft Access 2013 SQL-Syntax Information Dialog Box

the Microsoft Access database file name *Cape-Codd-SQL-92.accdb* is set to use Microsoft Access SQL-92. Choose the one you want to use (or use them both and compare the results!). Note that these files contain two additional tables (INVENTORY and WAREHOUSE) that we will not use in this chapter but that you will need for the Review Questions at the end of the chapter.

Alternatively, of course, you can create your own Microsoft Access database and then add the tables and data in Figures 2-3, 2-4, and 2-5, as described in Appendix A. If you create your own database, look at the Review Questions at the end of the chapter and create the INVENTORY and WAREHOUSE tables shown there in addition to the RETAIL_ORDER, ORDER_ITEM, and SKU tables shown in the chapter discussion. This will make sure that what you see on your monitor matches the screenshots in this chapter. Whether you download the database file or build it yourself, you will need to do one or the other before you can proceed.

Processing SQL Statements in Microsoft Access 2013

To process an SQL statement in Microsoft Access 2013, first open the database in Microsoft Access as described in Appendix A and then create a new tabbed Query window.

Opening a Microsoft Access Query Window in Design View

1. Click the **CREATE** command tab to display the Create command groups, as shown in Figure 2-8.
2. Click the **Query Design** button.
3. The Query1 tabbed document window is displayed in Design view, along with the Show Table dialog box, as shown in Figure 2-9.
4. Click the **Close** button on the Show Table dialog box. The Query1 document window now looks as shown in Figure 2-10. This window is used for creating and editing Microsoft Access queries in Design view and is used with Microsoft Access QBE.

Note that in Figure 2-10 the Select button is selected in the Query Type group on the Design tab. You can tell this is so because active or selected buttons are always shown in color on the Ribbon. This indicates that we are creating a query that is the equivalent of an SQL SELECT statement.

Also note that in Figure 2-10 the View gallery is available in the Results group of the Design tab. We can use this gallery to switch between Design view and SQL view. However, we can also just use the displayed SQL View button to switch to SQL view. The SQL View button is being displayed because Microsoft Access considers that to be the view you would most likely

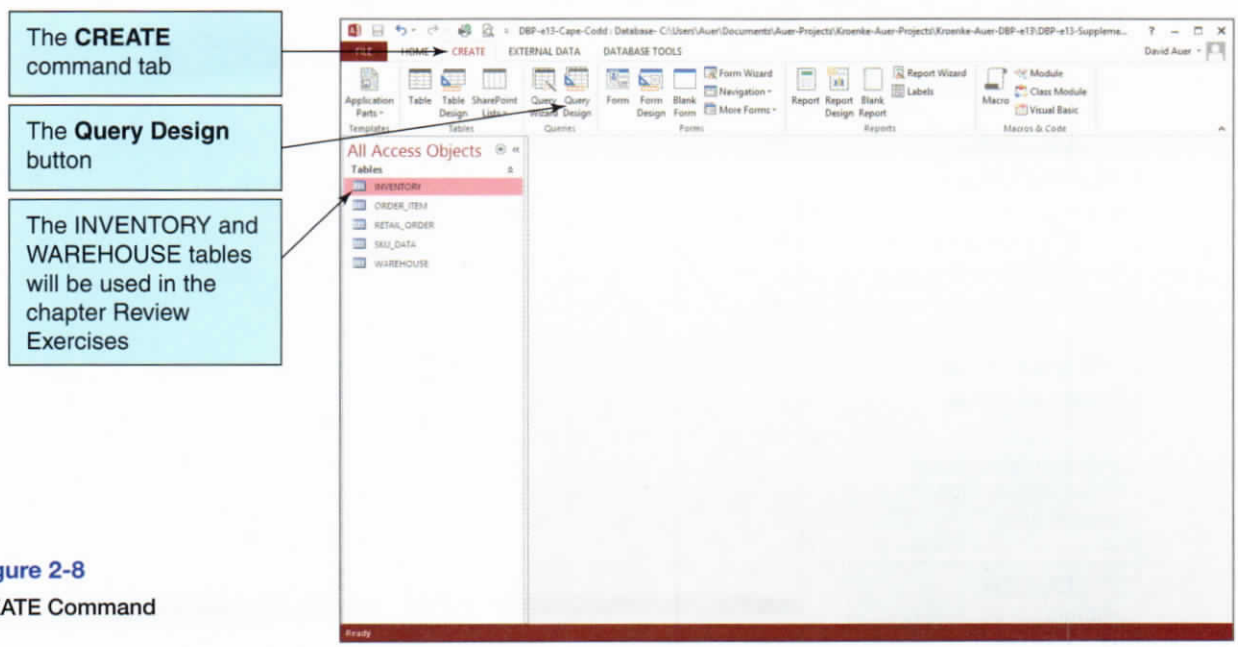


Figure 2-8
The CREATE Command Tab

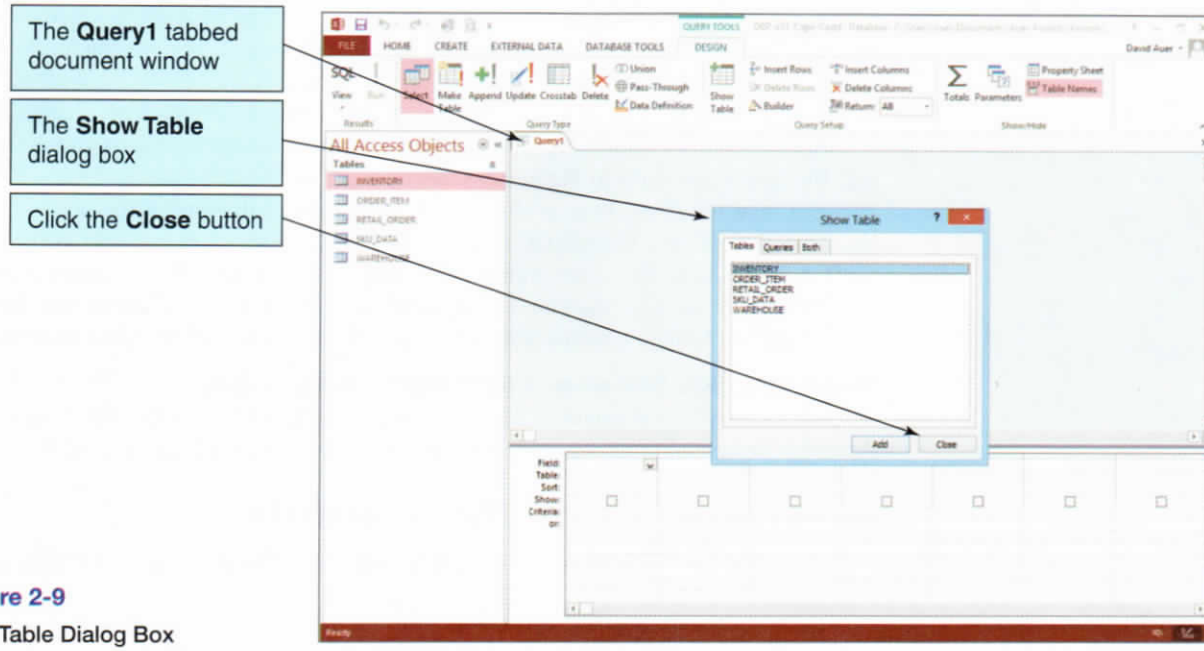


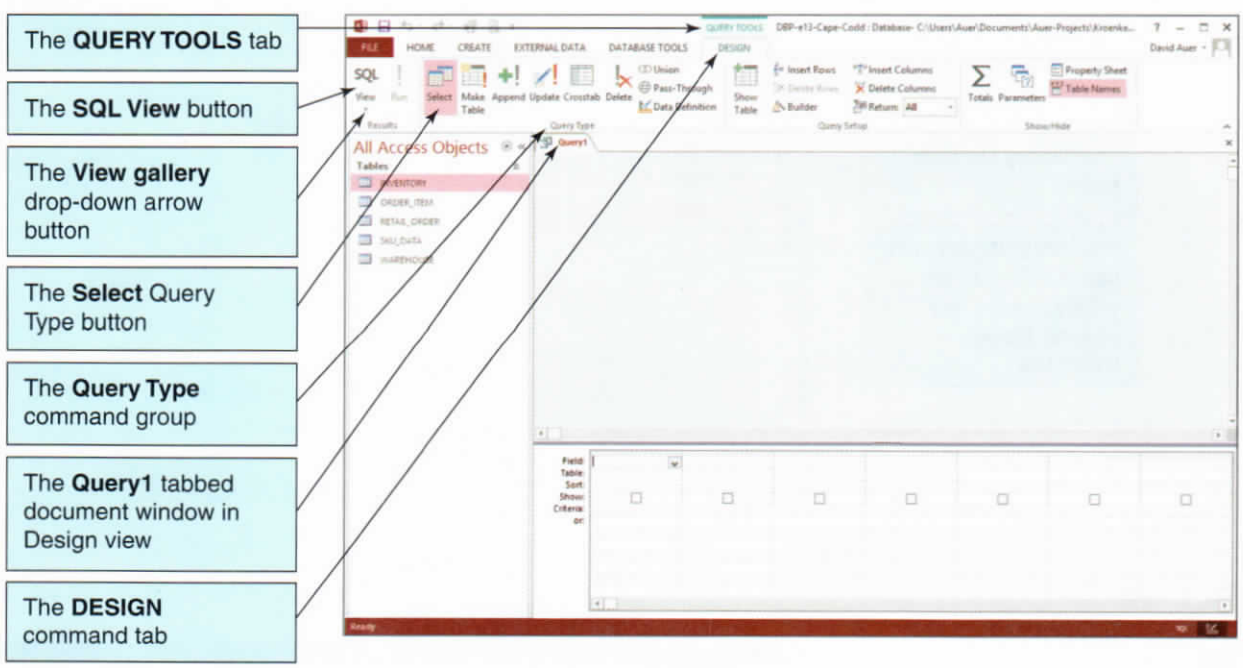
Figure 2-9
The Show Table Dialog Box

choose in the gallery if you used it. Microsoft Access always presents a "most likely needed" view choice as a button above the View gallery.

For our example SQL query in Microsoft Access, we will use SQL-Query-CH02-01, the first SQL query earlier in our discussion:

```
/* *** SQL-Query-CH02-01 *** */
SELECT     Department, Buyer
FROM       SKU_DATA;
```

Figure 2-10
The QUERY TOOLS Contextual Command Tab



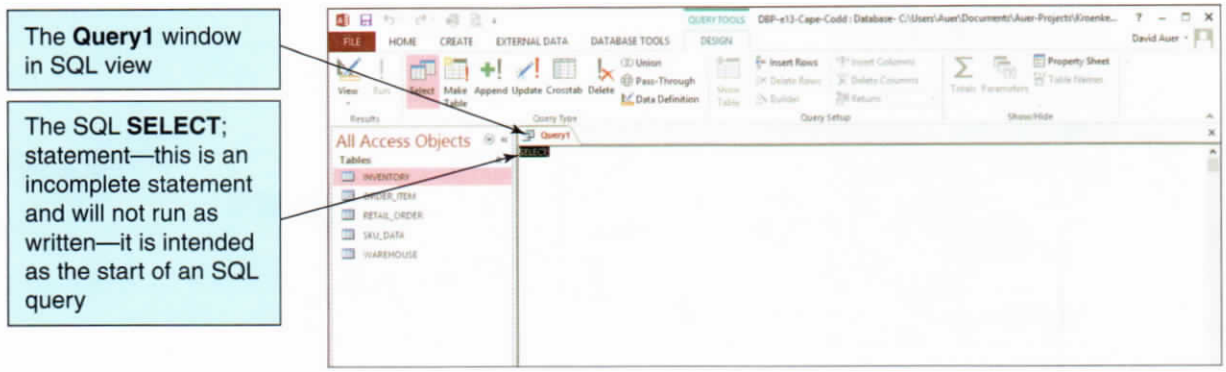


Figure 2-11
The Query1 Window in SQL View

Opening a Microsoft Access SQL Query Window and Running a Microsoft Access SQL Query

1. Click the **SQL View** button in the Results group on the Design tab. The Query1 window switches to the SQL view, as shown in Figure 2-11. Note the basic SQL command **SELECT;** that's shown in the window. This is an incomplete command, and running it will not produce any results.
2. Edit the SQL SELECT command to read (do *not* include the SQL comment line):

```
SELECT      Department, Buyer
FROM        SKU_DATA;
```

   as shown in Figure 2-12.
3. Click the **Run** button on the Design tab. The query results appear, as shown in Figure 2-13. Compare the results shown in Figure 2-13 to the SQL-Query-CH02-01 results shown on page 41.

Because Microsoft Access is a personal database and includes an application generator, we can save Microsoft Access queries for future use. Enterprise-level DBMS products generally do

Figure 2-12
The SQL Query

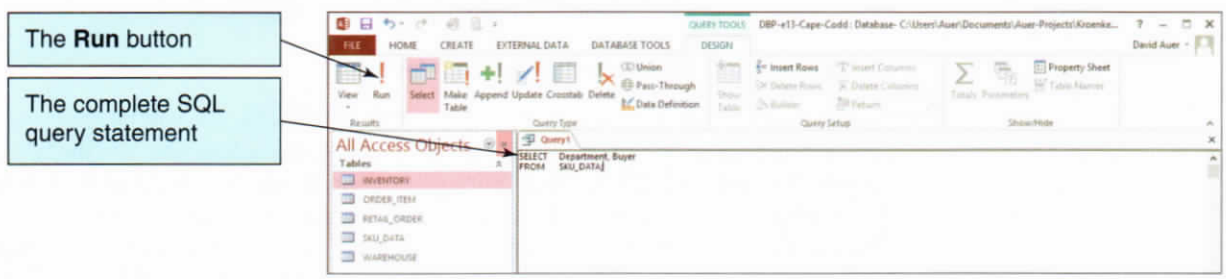
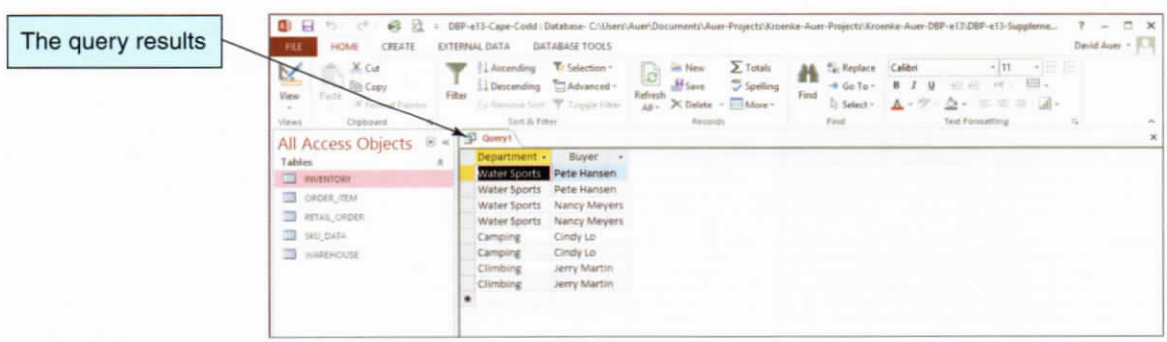


Figure 2-13
The SQL Query Results



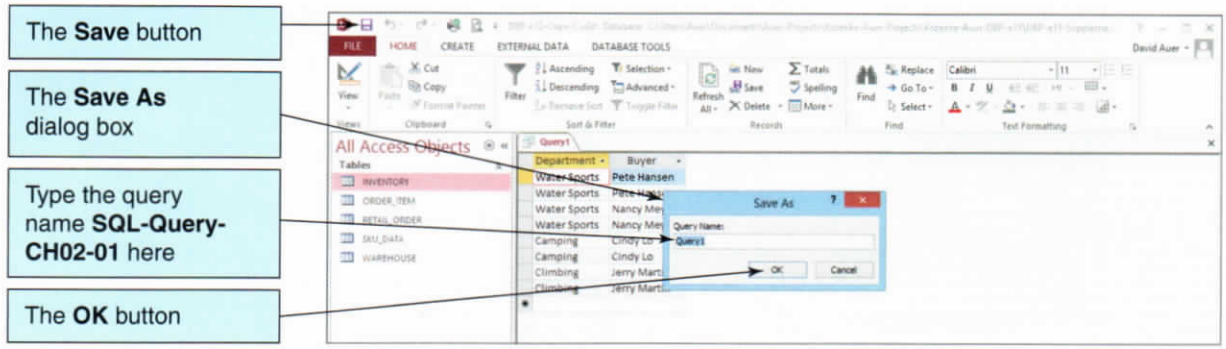


Figure 2-14
The Save As Dialog Box

not allow us to save queries (although they do allow us to save SQL Views within the database and SQL query scripts as separate files—we will discuss these methods later).

Saving a Microsoft Access SQL Query

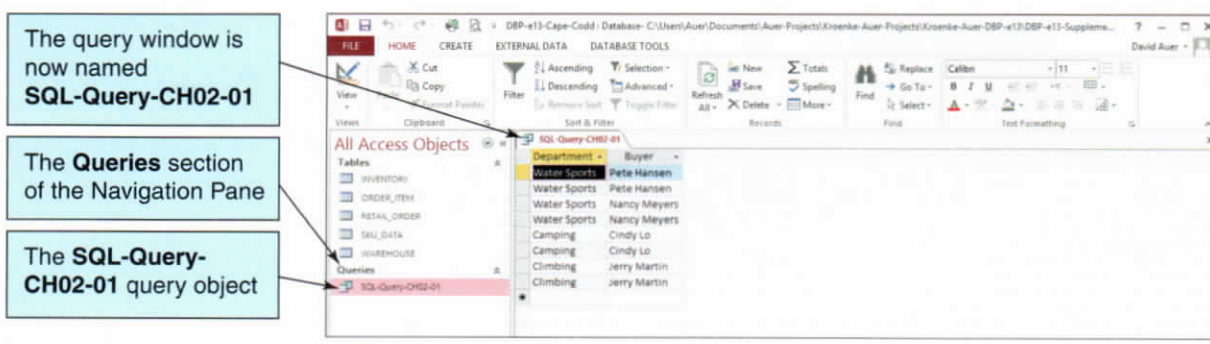
1. To save the query, click the **Save** button on the Quick Access Toolbar. The Save As dialog box appears, as shown in Figure 2-14.
2. Type in the query name **SQL-Query-CH02-01** and then click the **OK** button. The query is saved, and the window is renamed with the query name. As shown in Figure 2-15, the query document window is now named SQL-Query-CH02-01, and a newly created SQL-Query-CH02-01 query object appears in a Queries section of the Navigation Pane.
3. Close the SQL-Query-CH02-01 window by clicking the document window's **Close** button.
4. If Microsoft Access displays a dialog box asking whether you want to save changes to the design of the query SQL-Query-CH02-01, click the **Yes** button.

At this point, you should work through each of the other nine queries in the preceding discussion of the SQL SELECT/FROM/WHERE framework. Save each query as SQL-Query-CH02-##, where ## is a sequential number from 02 to 09 that corresponds to the SQL query label shown in the SQL comment line of each query.

Using SQL in Microsoft SQL Server 2012

Before you can use SQL statements with Microsoft SQL Server, you need access to a computer that has Microsoft SQL Server installed and that has a database with the tables and data shown in Figures 2-3, 2-4, and 2-5. Your instructor may have installed Microsoft SQL Server in your computer lab and entered the data for you. If so, follow his or her instructions for accessing that database. Otherwise, you will need to obtain a copy of Microsoft SQL Server 2012 and install it on your computer. Read the appropriate sections of Chapter 10A about obtaining and installing SQL Server 2012.

Figure 2-15
The Named and Saved Query



After you have Microsoft SQL Server 2012 installed, you will need to read the introductory discussion for using Microsoft SQL Server 2012 in Chapter 10A, starting on page 10A-1, and create the Cape Codd database. SQL Server scripts for creating and populating the Cape Codd database tables are available on our Web site at *www.pearsonhighered.com/kroenke*.

SQL Server 2012 uses the Microsoft SQL Server 2012 Management Studio as the GUI tool for managing the Microsoft SQL Server 2012 DBMS and the databases controlled by the DBMS. The Microsoft SQL Server 2012 Management Studio, which we will also refer to as just the SQL Server Management Studio, is installed as part of the Microsoft SQL Server 2012 installation process and is discussed in Chapter 10. Figure 2-16 shows the execution of SQL-Query-CH02-01 (note that the SQL comment is *not* included in the SQL statement as run—also note that the SQL comment *could* have been included in the SQL code if we had chosen to include it):

```
/* *** SQL-Query-CH02-01 *** */
SELECT     Department, Buyer
FROM       SKU_DATA;
```

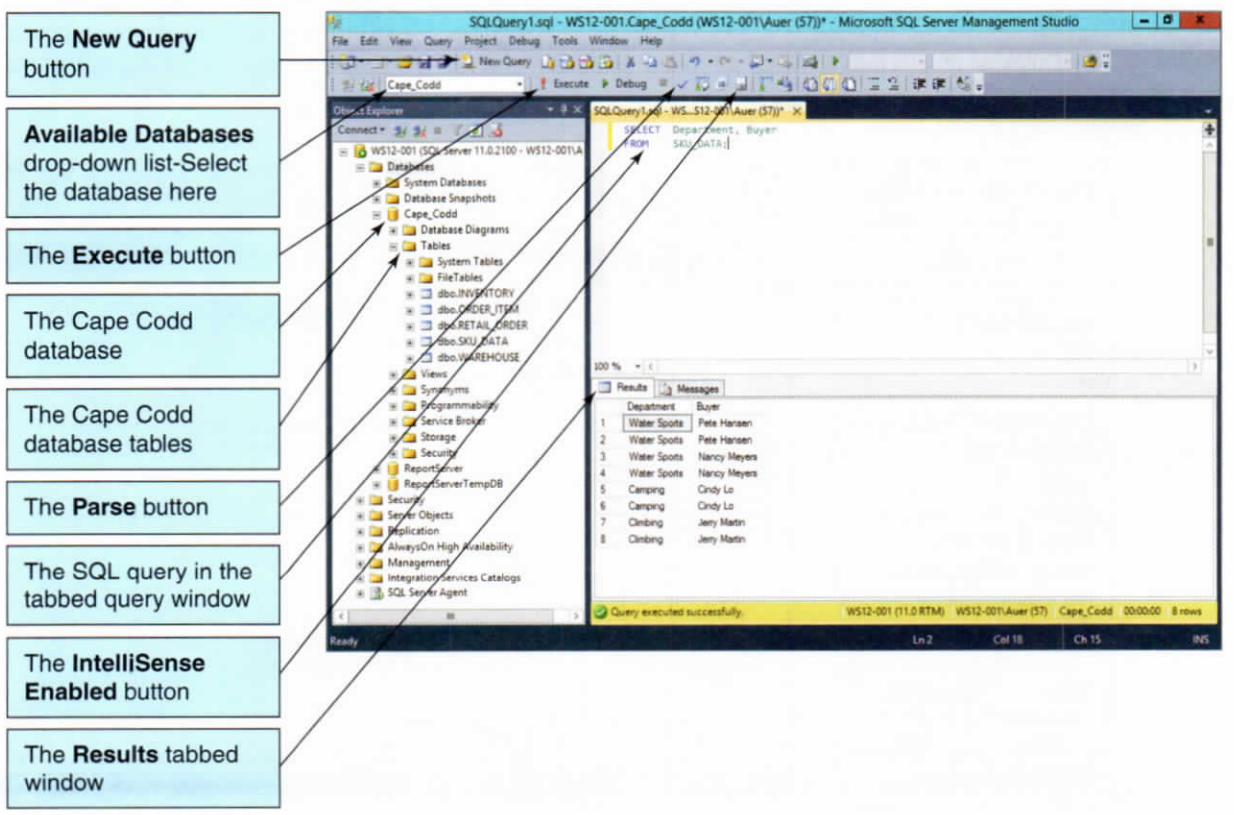
Running an SQL Query in Microsoft SQL Server Management Studio

1. Click the **New Query** button to display a new tabbed query window.
2. If the Cape Codd database is not displayed in the Available Database box, select it in the Available Databases drop-down list, and then click the **Intellisense Enabled** button to *disable* Intellisense.
3. Type the SQL SELECT command (*without* the SQL comment line shown above):

```
SELECT     Department, Buyer
FROM       SKU_DATA;
```

in the query window, as shown in Figure 2-16.

Figure 2-16
Running an SQL Query in Microsoft SQL Server Management Studio



4. At this point you can check the SQL command syntax before actually running the command by clicking the **Parse** button. A Results window will be displayed in the same location shown in Figure 2-16, but with the message "Command(s) completed successfully" if the SQL command syntax is correct or with an error message if there is a problem with the syntax.
5. Click the **Execute** button to run the query. The results are displayed in a results window, as shown in Figure 2-16.

Note that in Figure 2-16 the Cape Codd database object in the Object Browser in the left side window of the Microsoft SQL Server, Management Studio has been expanded to show the tables in the Cape Codd database. Many of the functions of the Microsoft SQL Server, Management Studio are associated with the objects in the Object Browser and are often accessed by right-clicking the object to display a shortcut menu.

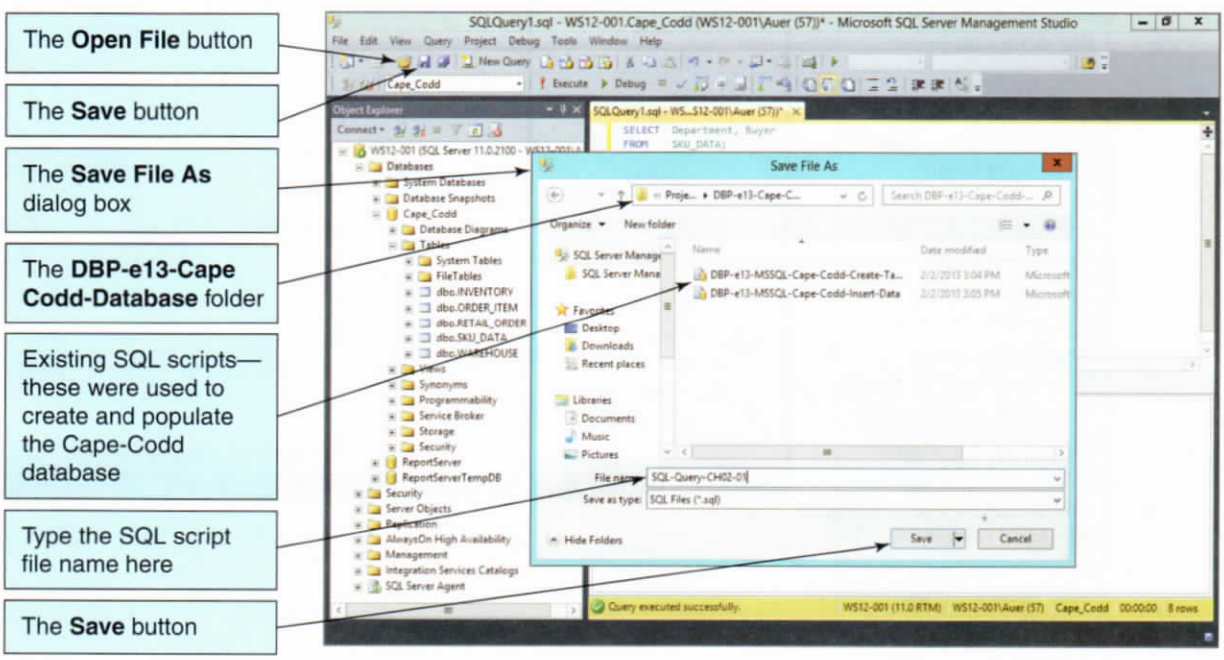
BY THE WAY

We are using Microsoft SQL Server 2012 Enterprise edition running in Microsoft Server 2012. When we give specific sequences of steps to follow in the text or figures in this book, we use the command terminology used by Microsoft SQL Server 2012 and associated utility programs in Microsoft Server 2012. If you are running a workstation operating system such as Microsoft XP or Microsoft Vista, the terminology may vary somewhat.

Microsoft SQL Server 2012 is an enterprise-class DBMS product and, as is typical of such products, does not store queries within the DBMS (it does store SQL Views, which can be considered a type of query, and we will discuss SQL Views in Chapter 7). However, you can save queries as SQL script files. An **SQL script file** is a separately stored plain text file, and it usually uses a file name extension of **.sql*. An SQL script can be opened and run as an SQL command (or set of commands). Often used to create and populate databases, scripts can also be used to store a query or set of queries. Figure 2-17 shows the SQL query being saved as an SQL script.

Note that in Figure 2-17 the SQL scripts are shown in a folder named *DBP-e13-Cape-Codd-Database*. When the Microsoft SQL Server 2012 Management Studio is installed, a new folder named *SQL Server Management Studio* is created in your My Documents folder. Within this folder, create a *Projects* subfolder, and use the Projects folder as the location for storing your SQL script files.

Figure 2-17
Saving an SQL Query as an SQL Script in Microsoft SQL Server Management Studio



We recommend that you create a folder for each database in the Projects folder. We have created a folder named *DBC-e13-Cape-Codd-Database* to store the script files associated with the Cape Codd database.

Saving a Microsoft SQL Server Query as an SQL Script in Microsoft SQL Server Management Studio

1. Click the **Save** button shown in Figure 2-17. The Save File As dialog appears, as shown in Figure 2-17.
2. Browse to the *\My Documents\SQL Server Management Studio\Projects\DBP-e13-Cape-Codd-Database* folder.
3. Note that there are already two SQL script names displayed in the dialog box. These are the SQL scripts that were used to create and populate the Cape Codd database tables, and they are available on our Web site at *www.pearsonhighered.com/kroenke.*
4. In the File Name text box, type the SQL script file name *SQL-Query-CH02-01.*
5. Click the **Save** button.

To rerun the saved query, you would click the **Open File** button shown in Figure 2-17 to open the Open File dialog box, open the query, and then click the **Execute** button.

At this point, you should work through each of the other nine queries in the preceding discussion of the SQL SELECT/FROM/WHERE framework. Save each query as SQL-Query-CH02-##, where ## is a sequential number from 02 to 09 that corresponds to the SQL query label shown in the SQL comment line of each query.

Using SQL in Oracle Database 11*g* Release 2

Before you can enter SQL statements into Oracle Database 11*g* Release 2 you need access to a computer that has Oracle Database 11*g* Release 2 installed and that has a database with the tables and data shown in Figures 2-3, 2-4, and 2-5. Your instructor may have installed Oracle Database 11*g* Release 2 on a computer in the lab and entered the data for you. If so, follow his or her instructions for accessing that database. Otherwise, you will need to obtain a copy of Oracle Database 11*g* Release 2 and install it on your computer. Read the appropriate sections of Chapter 10B about obtaining and installing Oracle Database 11*g* Release 2.

After you have installed Oracle Database 11*g* Release 2, you will need to read the introductory discussion for Oracle Database 11*g* Release 2 in Chapter 10B, starting on page 10B-1, and create the Cape Codd database. Oracle Database 11*g* Release 2 scripts for creating and populating the Cape Codd database tables are available on our Web site at *www.pearsonhighered.com/kroenke.*

Although Oracle users have been dedicated to the Oracle SQL*Plus command line tool, professionals are moving to the new Oracle SQL Developer GUI tool. This application is installed as part of the Oracle Database 11*g* Release 2 installation, and updated versions are available for free download at *www.oracle.com/technology/software/products/sql/index.html.* We will use it as our standard GUI tool for managing the databases created by the Oracle Database DBMS. Figure 2-18 shows the execution of SQL-Query-CH02-01 (note that the SQL comment is *not* included in the SQL statement as run—also note that the SQL comment *could* have been included in the SQL code if we had chosen to include it):

```
/* *** SQL-Query-CH02-01 *** */
SELECT     Department, Buyer
FROM       SKU_DATA;
```

Running an SQL Query in Oracle SQL Developer

1. Click the **New Connection** button and open the Cape Codd database.
2. In the tabbed SQL Worksheet, type the SQL SELECT command (*without* the SQL comment line shown above):

```
SELECT     Department, Buyer
FROM       SKU_DATA;
```

   as shown in Figure 2-18.

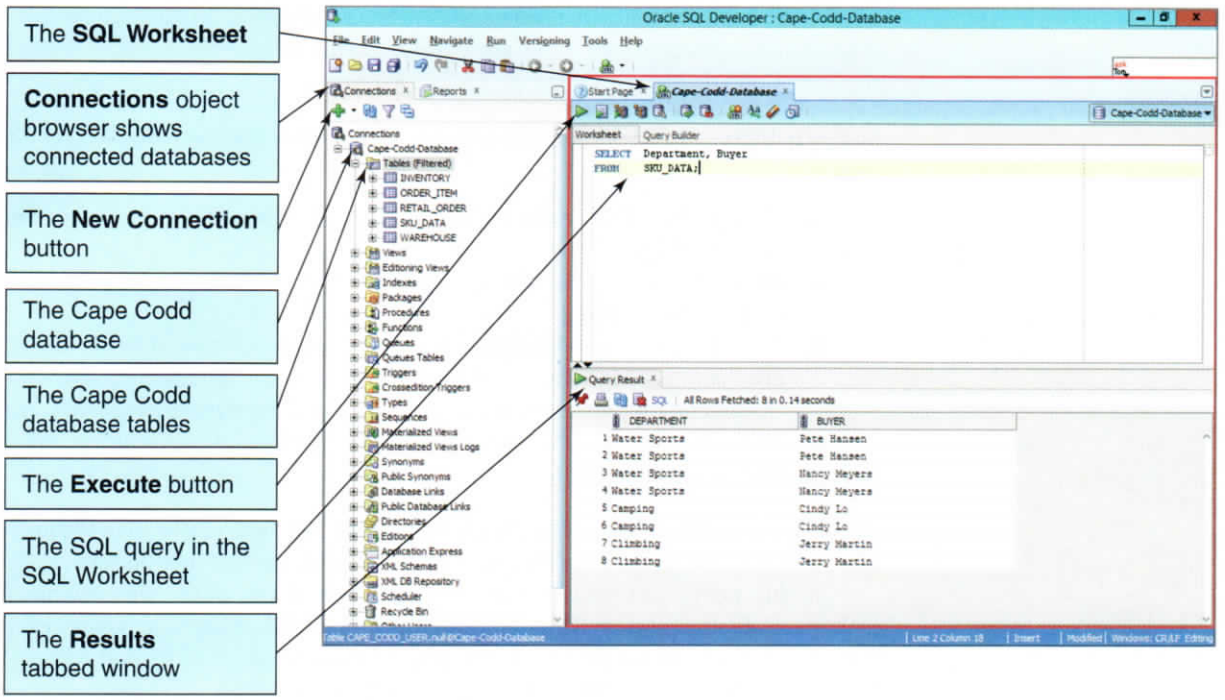


Figure 2-18
Running an SQL Query in Oracle SQL Developer

3. Click the **Execute** button to run the query. The results are displayed in a results window, as shown in Figure 2-18.

Note that in Figure 2-18, the Cape-Codd-Database object in the Object Browser in the left side Connection object browser of the Oracle SQL Developer has been expanded to show the tables in the Cape Codd database. Many of the functions of SQL Developer are associated with the objects in the Connections object browser and are often accessed by right-clicking the object to display a shortcut menu.

BY THE WAY

We are using Oracle Database 11*g* Release 2 running in Microsoft Server 2008 R2. When we give specific sequences of steps to follow in the text or figures in this book, we use the command terminology used by Oracle Database 11*g* Release 2 and associated utility programs in Microsoft Server 2008 R2. If you are running a workstation operating system such as Microsoft XP, Microsoft Vista, or Linux, the terminology may vary somewhat.

Oracle Database 11*g* Release 2 is an enterprise-class DBMS product and, as is typical of such products, does not store queries within the DBMS (it does store SQL Views, which can be considered a type of query, and we will discuss SQL Views later in this chapter). However, you can save queries as SQL script files. An **SQL script file** is a separately stored plain text file, and it usually has a file name extension of **.sql*. An SQL script can be opened and run as an SQL command (or set of commands). Often used to create and populate databases, scripts can also be used to store a query or set of queries. Figure 2-19 shows the SQL query being saved as an SQL script.

Note that in Figure 2-19 the SQL scripts are shown in a folder named *{UserName}\Documents\SQL Developer\DBP-e13-Cape-Codd-Database*. By default, Oracle SQL Developer stores **.sql* files in an obscure location within its own application files. We recommend that you create a subfolder in your My Documents folder named *SQL Developer* and then create a

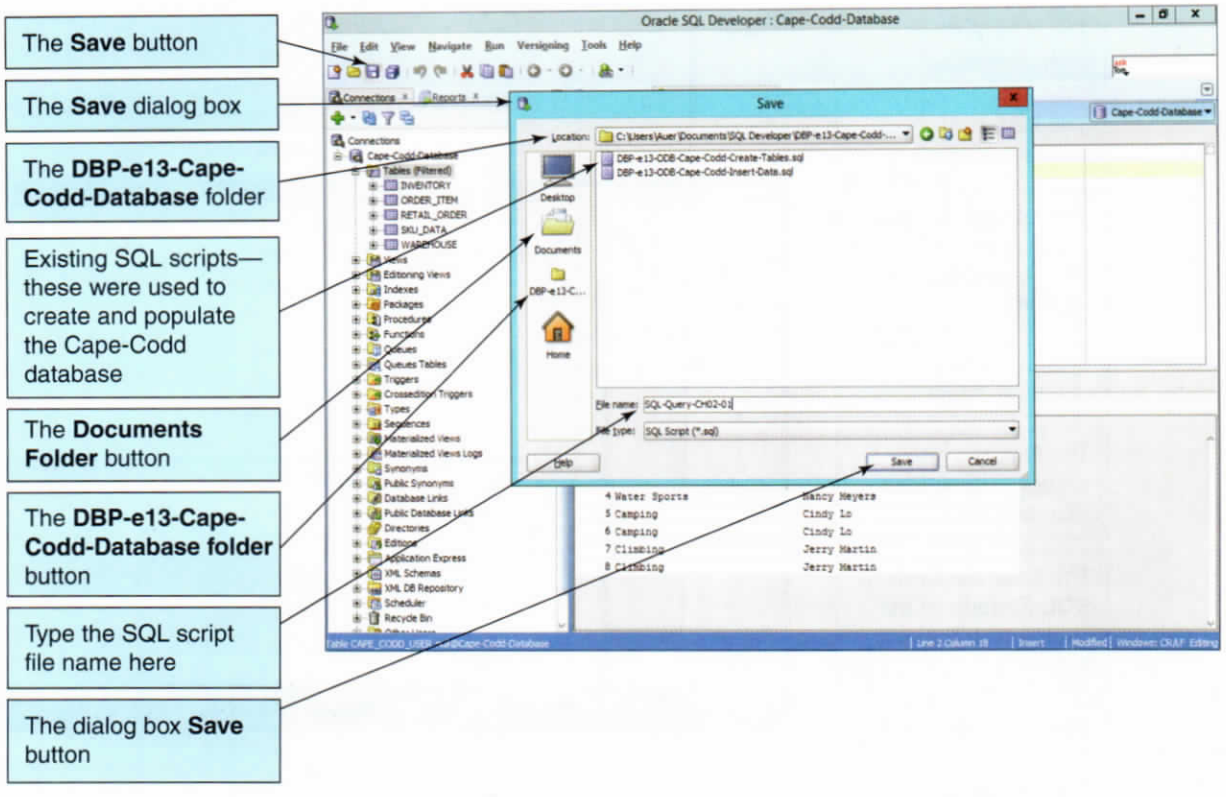


Figure 2-19
Saving an Oracle SQL Query as an SQL Script in Oracle SQL Developer

subfolder for each database in the SQL Developer folder. We have created a folder named *DBP-e13-Cape-Codd-Database.* to store the script files associated with the Cape Codd database.

Saving an SQL Script in Oracle SQL Developer

1. Click the **Save** button shown in Figure 2-19. The Save dialog appears, as shown in Figure 2-19.
2. Click the Documents button on the Save dialog box to move to the Documents folder and then browse to the DBP-e13-Cape-Codd-Database folder.
3. Note that there are already two SQL script names displayed in the dialog box. These are the SQL scripts that were used to create and populate the Cape Codd database tables, and they are available on our Web site at *www.pearsonhighered.com/kroenke.*
4. In the File Name text box, type the SQL script file name *SQL-Query-CH02-01.sql.*
5. Click the **Save** button.

To rerun the saved query, you would click the SQL Developer **Open File** button to open the Open File dialog box, browse to the query file, open the query file, and then click the **Execute** button.

At this point, you should work through each of the other nine queries in the preceding discussion of the SQL SELECT/FROM/WHERE framework. Save each query as SQLQuery-CH02-##, where ## is a sequential number from 02 to 09 that corresponds to the SQL query label shown in the SQL comment line of each query.

Using SQL in Oracle MySQL 5.6

Before you can use SQL statements with Oracle MySQL 5.6, you need access to a computer that has MySQL 5.6 installed and that has a database with the tables and data shown in Figure 2-4, 2-5, and 2-6. Your instructor may have installed MySQL 5.6 in your computer lab and entered the data for you. If so, follow his or her instructions for accessing that database. Otherwise, you will need to obtain a copy of MySQL Community Server 5.6 and install

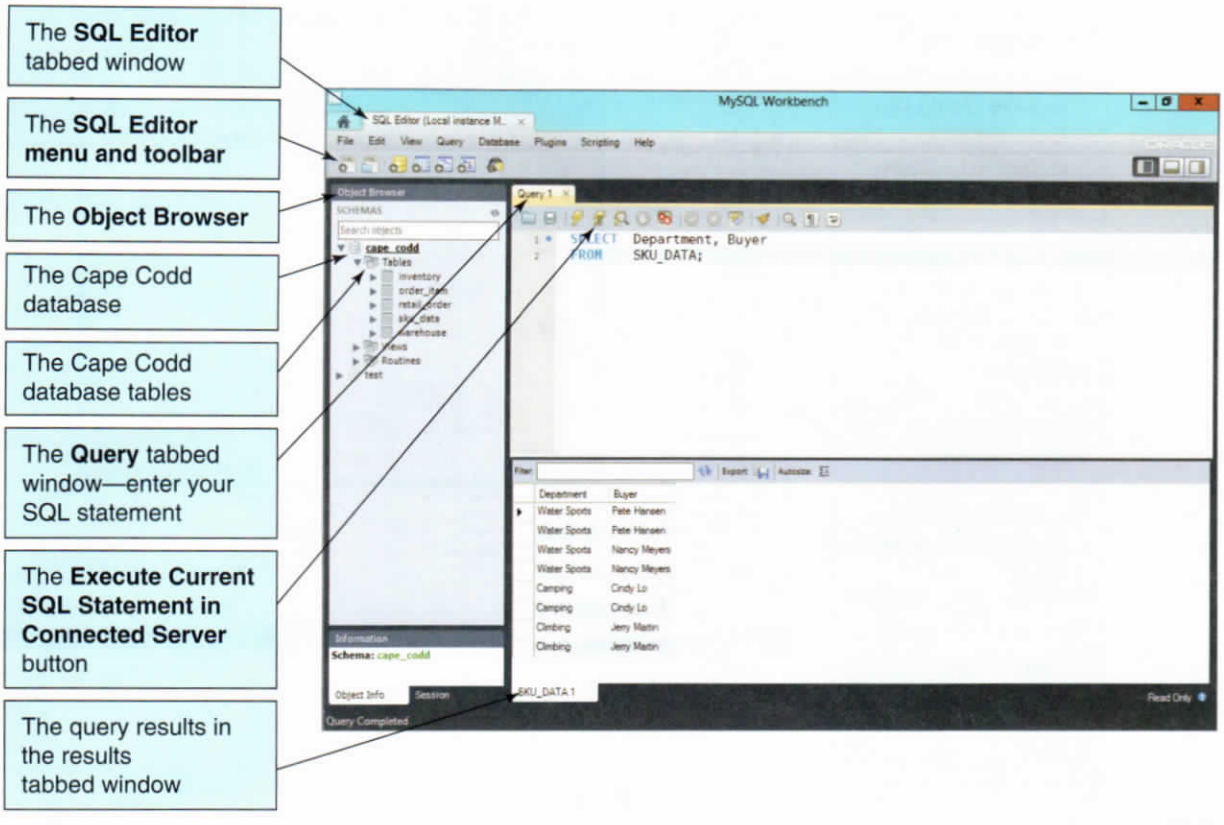


Figure 2-20
Running an SQL Query in the MySQL Workbench

it on your computer. Read the appropriate sections of Chapter 10C about obtaining and installing MySQL Community Server 5.6.

After you have MySQL 5.6 installed, you will need to read the introductory discussion for MySQL 5.6 in Chapter 10C, starting on page 10C-1, and create the Cape Codd database. MySQL scripts for creating and populating the Cape Codd database tables are available on our Web site at *www.pearsonhighered.com/kroenke*.

MySQL 5.6 uses the MySQL Workbench as the GUI tool for managing the MySQL 5.6 DBMS and the databases controlled by the DBMS. This tool must be installed separately from the MySQL 5.6 DBMS and this is discussed in Chapter 10C. SQL statements are created and run in the MySQL Workbench, and Figure 2-20 shows the execution of SQL-Query-CH02-01 (note that the SQL comment is *not* included in the SQL statement as run—also note that the SQL comment *could* have been included in the SQL code if we had chosen to include it):

```
/* *** SQL-Query-CH02-01 *** */
SELECT     Department, Buyer
FROM       SKU_DATA;
```

Running an SQL Query in the MySQL Workbench

1. To make the Cape Codd database the default schema (active database), right-click the cape_codd schema (database) object to display the shortcut menu and then click the **Set as Default Schema** command.
2. In the Query 1 tabbed window in the SQL Editor tabbed window, type the SQL SELECT command (*without* the SQL comment line shown above):

```
SELECT     Department, Buyer
FROM       SKU_DATA;
```

as shown in Figure 2-20.

3. Click the **Execute Current SQL Statement in Connected Server** button to run the query. The results are displayed in a tabbed Query Result window, shown as the Query 1 Result window in Figure 2-20 (you can have more than one Query Result window open, and thus they need to be numbered).

Note that in Figure 2-20 the Cape Codd database object in the Object Browser in the left-side window of the MySQL Workbench has been expanded to show the tables in the Cape Codd database. Many of the functions of the MySQL Workbench are associated with the objects in the Object Browser and are often accessed by right-clicking the object to display a shortcut menu.

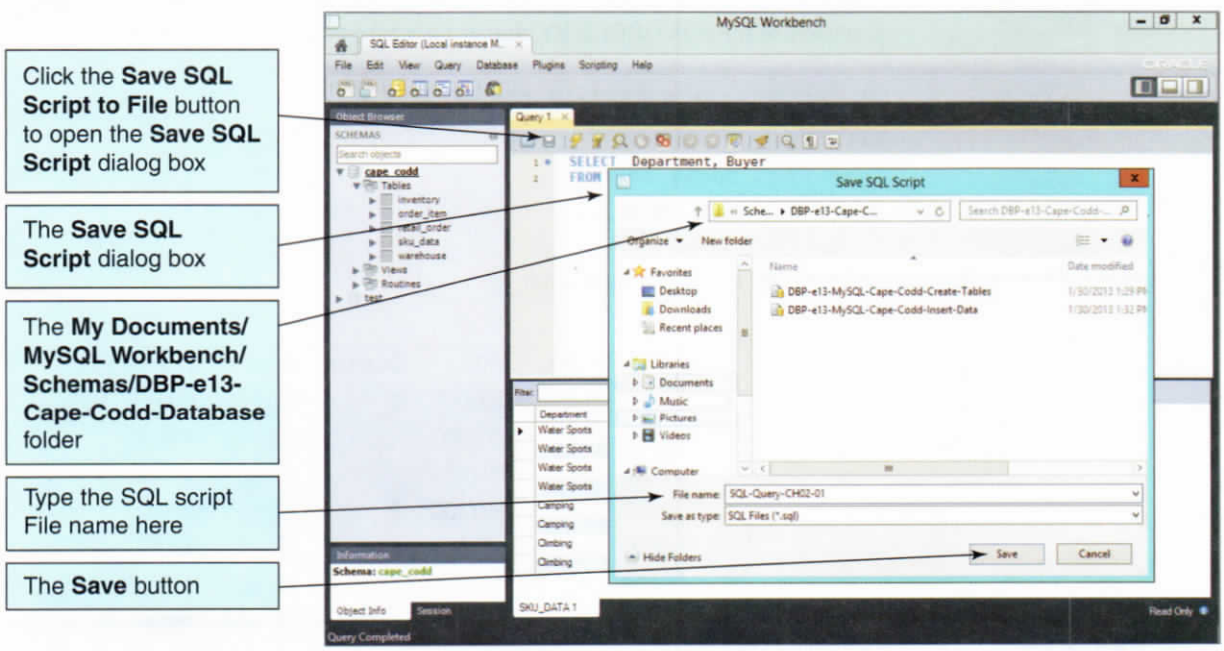
BY THE WAY

We are using MySQL Community Server 5.6 running in Microsoft Server 2012. When we give specific sequences of steps to follow in the text or figures in this book, we use the command terminology used for MySQL 5.6 and associated utility programs in Microsoft Server 2012. If you are running a workstation operating system such as Microsoft XP, Microsoft Vista, or Linux, the terminology may vary somewhat.

MySQL 5.6 is an enterprise-class DBMS product and, as is typical of such products, does not store queries within the DBMS (it does store SQL Views, which can be considered a type of query, and we will discuss SQL Views later in this chapter). However, you can save MySQL queries as SQL script files. An **SQL script file** is a separately stored plain text file, and it usually uses a file name extension of **.sql*. An SQL script file can be opened and run as an SQL command. Figure 2-21 shows the SQL query being saved as an SQL script file.

Note that in Figure 2-21 the query will be saved in a folder named *My Documents\MySQL Workbench\Schemas\DBP-e13-Cape-Codd-Database*. By default, MySQL Workbench stores files in the user's My Documents folder. We recommend that you create a subfolder in your My Documents folder named *MySQL Workbench* and then create subfolders labeled *EER Models* and *Schemas*. Within each of these subfolders, create a sub-subfolder for each MySQL 5.6 database. We have created a folder named *DBP-e13-Cape-Codd-Database* to store the script files associated with the Cape Codd database.

Figure 2-21
Saving a MySQL Query as an SQL Script in the MySQL Workbench



Saving a MySQL 5.6 Query

1. Click the **Save SQL Script to File** button, as shown in Figure 2-21. The Save Query to File dialog appears, as shown in Figure 2-21.
2. Browse to the *My Documents\MySQL Workbench\Schemas\DBP-e13-Cape-Codd-Database* folder.
3. In the File name text box, type the SQL query file name *SQL-Query-CH02-01*.
4. Click the **Save** button.

To rerun the saved query, you would click the **File | Open SQL Script** menu command to open the **Open SQL Script** dialog box, then select and open the SQL query *.sql files, and, finally, click the **Execute Current SQL Statement in Connected Server** button.

At this point, you should work through each of the other nine queries in the preceding discussion of the SQL SELECT/FROM/WHERE framework. Save each query as SQLQuery-CH02-##, where ## is a sequential number from 02 to 09 that corresponds to the SQL query label shown in the SQL comment line of each query.

SQL Enhancements for Querying a Single Table

We started our discussion of SQL queries with SQL statements for processing a single table, and now we will add an additional SQL feature to those queries. As we proceed, you will begin to see how powerful SQL can be for querying databases and for creating information from existing data.

BY THE WAY

The SQL results shown in this chapter were generated using Microsoft SQL Server 2012. Query results from other DBMS products will be similar, but may vary a bit.

Sorting the SQL Query Results

The order of the rows produced by an SQL statement is arbitrary and determined by programs in the bowels of each DBMS. If you want the DBMS to display the rows in a particular order, you can use the **SQL ORDER BY clause**. For example, the SQL statement:

```
/* *** SQL-Query-CH02-10 *** */
SELECT      *
FROM        ORDER_ITEM
ORDER BY    OrderNumber;
```

will generate the following results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	1000	202000	1	130.00	130.00
3	2000	101100	4	50.00	200.00
4	2000	101200	2	50.00	100.00
5	3000	101200	1	50.00	50.00
6	3000	101100	2	50.00	100.00
7	3000	100200	1	300.00	300.00

We can sort by two columns by adding a second column name. For example, to sort first by OrderNumber and then by Price within OrderNumber, we use the following SQL query:

```
/* *** SQL-Query-CH02-11 *** */
SELECT      *
FROM        ORDER_ITEM
ORDER BY    OrderNumber, Price;
```

The result for this query is:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	202000	1	130.00	130.00
2	1000	201000	1	300.00	300.00
3	2000	101100	4	50.00	200.00
4	2000	101200	2	50.00	100.00
5	3000	101200	1	50.00	50.00
6	3000	101100	2	50.00	100.00
7	3000	100200	1	300.00	300.00

If we want to sort the data by Price and then by OrderNumber, we would simply reverse the order of those columns in the ORDER BY clause as follows:

```
/* *** SQL-Query-CH02-12 *** */
SELECT      *
FROM        ORDER_ITEM
ORDER BY    Price, OrderNumber;
```

with the results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	2000	101100	4	50.00	200.00
2	2000	101200	2	50.00	100.00
3	3000	101200	1	50.00	50.00
4	3000	101100	2	50.00	100.00
5	1000	202000	1	130.00	130.00
6	1000	201000	1	300.00	300.00
7	3000	100200	1	300.00	300.00

Note to Microsoft Access users: Unlike the Microsoft SQL Server output shown here, Microsoft Access displays dollar signs in the output of currency data.

By default, rows are sorted in ascending order. To sort in descending order, add the **SQL DESC keyword** after the column name. Thus, to sort first by Price in descending order and then by OrderNumber in ascending order, we use the SQL query:

```
/* *** SQL-Query-CH02-13 *** */
SELECT      *
FROM        ORDER_ITEM
ORDER BY    Price DESC, OrderNumber ASC;
```

The result is:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	3000	100200	1	300.00	300.00
3	1000	202000	1	130.00	130.00
4	2000	101100	4	50.00	200.00
5	2000	101200	2	50.00	100.00
6	3000	101200	1	50.00	50.00
7	3000	101100	2	50.00	100.00

Because the default order is ascending, it is not necessary to specify ASC in the last SQL statement. Thus, the following SQL statement is equivalent to the previous SQL query:

```
/* *** SQL-Query-CH02-14 *** */
SELECT      *
FROM        ORDER_ITEM
ORDER BY    Price DESC, OrderNumber;
```

and produces the same results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	201000	1	300.00	300.00
2	3000	100200	1	300.00	300.00
3	1000	202000	1	130.00	130.00
4	2000	101100	4	50.00	200.00
5	2000	101200	2	50.00	100.00
6	3000	101200	1	50.00	50.00
7	3000	101100	2	50.00	100.00

SQL WHERE Clause Options

SQL includes a number of SQL WHERE clause options that greatly expand SQL's power and utility. In this section, we consider three options: compound clauses, ranges, and wildcards.

Compound WHERE Clauses

SQL WHERE clauses can include multiple conditions by using the SQL AND, OR, IN, and NOT IN operators. For example, to find all of the rows in SKU_DATA that have a Department named Water Sports and a Buyer named Nancy Meyers, we can use the **SQL AND operator** in our query code:

```
/* *** SQL-Query-CH02-15 *** */
SELECT      *
FROM        SKU_DATA
WHERE       Department='Water Sports'
    AND     Buyer='Nancy Meyers';
```

The results of this query are:

	SKU	SKU_Description	Department	Buyer
1	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers

Similarly, to find all of the rows of SKU_DATA for either the Camping or Climbing departments, we can use the **SQL OR operator** in the SQL query:

```
/* *** SQL-Query-CH02-16 *** */
SELECT      *
FROM        SKU_DATA
WHERE       Department='Camping'
    OR      Department='Climbing';
```

which gives us the following results:

	SKU	SKU_Description	Department	Buyer
1	201000	Half-dome Tent	Camping	Cindy Lo
2	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
3	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
4	302000	Locking Carabiner, Oval	Climbing	Jerry Martin

Three or more AND and OR conditions can be combined, but in such cases the **SQL IN operator** and the **SQL NOT IN operator** are easier to use. For example, suppose we want to obtain all of the rows in SKU_DATA for buyers Nancy Meyers, Cindy Lo, and Jerry Martin. We could construct a WHERE clause with two ANDs, but an easier way to do this is to use the IN operator, as illustrated in the SQL query:

```
/* *** SQL-Query-CH02-17 *** */
SELECT      *
FROM        SKU_DATA
WHERE       Buyer IN ('Nancy Meyers', 'Cindy Lo', 'Jerry Martin');
```

In this format, a set of values is enclosed in parentheses. A row is selected if Buyer is equal to any one of the values provided. The result is:

	SKU	SKU_Description	Department	Buyer
1	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
3	201000	Half-dome Tent	Camping	Cindy Lo
4	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
5	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
6	302000	Locking Carabiner, Oval	Climbing	Jerry Martin

Similarly, if we want to find rows of SKU_DATA for which the buyer is someone *other* than Nancy Meyers, Cindy Lo, or Jerry Martin, we would use the SQL query:

```
/* *** SQL-Query-CH02-18 *** */
SELECT      *
FROM        SKU_DATA
WHERE       Buyer NOT IN ('Nancy Meyers', 'Cindy Lo', 'Jerry Martin');
```

The result is:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen

Observe an important difference between IN and NOT IN. A row qualifies for an IN condition if the column is equal to *any* of the values in the parentheses. However, a row qualifies for a NOT IN condition if it is not equal to *all* of the items in the parentheses.

Ranges in SQL WHERE Clauses

SQL WHERE clauses can specify ranges of data values by using the **SQL BETWEEN keyword**. For example, the following SQL statement:

```
/* *** SQL-Query-CH02-19 *** */
SELECT      *
FROM        ORDER_ITEM
WHERE       ExtendedPrice BETWEEN 100 AND 200;
```

will produce the following results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	2000	101100	4	50.00	200.00
2	3000	101100	2	50.00	100.00
3	2000	101200	2	50.00	100.00
4	1000	202000	1	130.00	130.00

Notice that both the ends of the range, 100 and 200, are included in the resulting table. The preceding SQL statement is equivalent to the SQL query:

```
/* *** SQL-Query-CH02-20 *** */
SELECT      *
FROM        ORDER_ITEM
WHERE       ExtendedPrice >= 100
    AND     ExtendedPrice <= 200;
```

And which, of course, produces identical results:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	2000	101100	4	50.00	200.00
2	3000	101100	2	50.00	100.00
3	2000	101200	2	50.00	100.00
4	1000	202000	1	130.00	130.00

Wildcards in SQL WHERE Clauses

The **SQL LIKE keyword** can be used in SQL WHERE clauses to specify matches on portions of column values. For example, suppose we want to find the rows in the SKU_DATA table for all buyers whose first name is *Pete*. To find such rows, we use the SQL keyword LIKE with the **SQL percent sign (%) wildcard character**, as shown in the SQL query:

```
/* *** SQL-Query-CH02-21 *** */
SELECT      *
FROM        SKU_DATA
WHERE       Buyer LIKE 'Pete%';
```

When used as an SQL wildcard character, the percent symbol (%) stands for any sequence of characters. When used with the SQL LIKE keyword, the character string 'Pete%' means any sequence of characters that starts with the letters *Pete*. The result of this query is:

	SKU	SKU_Description	Department	Buyer
1	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen

Does Not Work with Microsoft Access ANSI-89 SQL

Microsoft Access ANSI-89 SQL uses wildcards, but not the SQL-92 standard wildcards. Microsoft Access uses the Microsoft Access asterisk (*) wildcard character instead of a percent sign to represent multiple characters.

Solution: Use the Microsoft Access asterisk (*) wildcard in place of the SQL-92 percent sign (%) wildcard in Microsoft Access ANSI-89 SQL statements. Thus, the preceding SQL query would be written as follows for Microsoft Access:

```
/* *** SQL-Query-CH02-21-Access *** */
SELECT      *
FROM        SKU_DATA
WHERE       Buyer LIKE 'Pete*';
```

Suppose we want to find the rows in SKU_DATA for which the SKU_Description includes the word *Tent* somewhere in the description. Because the word *Tent* could be at the front, at the end, or in the middle, we need to place a wildcard on both ends of the LIKE phrase, as follows:

```
/* *** SQL-Query-CH02-22 *** */
SELECT      *
FROM        SKU_DATA
WHERE       SKU_Description LIKE '%Tent%';
```

This query will find rows in which the word *Tent* occurs in any place in the SKU_Description. The result is:

	SKU	SKU_Description	Department	Buyer
1	201000	Half-dome Tent	Camping	Cindy Lo
2	202000	Half-dome Tent Vestibule	Camping	Cindy Lo

Sometimes we need to search for a particular value in a particular location in the column. For example, assume SKU values are coded such that a 2 in the third position from the right has some particular significance; maybe it means that the product is a variation of another product. For whatever reason, assume that we need to find all SKUs that have a 2 in the third column from the right. Suppose we try the SQL query:

```
/* *** SQL-Query-CH02-23 *** */
SELECT      *
FROM        SKU_DATA
WHERE       SKU LIKE '%2%';
```

The result is:

	SKU	SKU_Description	Department	Buyer
1	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
3	201000	Half-dome Tent	Camping	Cindy Lo
4	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
5	302000	Locking Carabiner, Oval	Climbing	Jerry Martin

This is *not* what we wanted. We mistakenly retrieved all rows that had a 2 in *any* position in the value of SKU. To find the products we want, we cannot use the SQL wildcard character %. Instead, we must use the **SQL underscore (_) wildcard character**, which represents a single, unspecified character in a specific position. The following SQL statement will find all SKU_DATA rows with a value of 2 in the third position from the right:

```
/* *** SQL-Query-CH02-24 *** */
SELECT      *
FROM        SKU_DATA
WHERE       SKU LIKE '%2__';
```

Observe that there are *two* underscores in this SQL query—one for the first position on the right and another for the second position on the right. This query gives us the result that we want:

	SKU	SKU_Description	Department	Buyer
1	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers

Does Not Work with Microsoft Access ANSI-89 SQL

Microsoft Access ANSI-89 SQL uses wildcards, but not the SQL-92 standard wildcards. Microsoft Access uses the **Microsoft Access question mark (?) wildcard character** instead of an underscore (_) to represent a single character.

Solution: Use the Microsoft Access question mark (?) wildcard in place of the SQL-92 underscore (_) wildcard in Microsoft Access ANSI-89 SQL statements. Thus, the preceding SQL query would be written as follows for Microsoft Access:

```
/* *** SQL-Query-CH02-24-Access *** */
SELECT      *
FROM        SKU_DATA
WHERE       SKU LIKE '*2??';
```

Furthermore, Microsoft Access can sometimes be fussy about stored trailing spaces in a text field. You may have problems with a WHERE clause like this:

```
WHERE       SKU LIKE '10?200';
```

Solution: Use a trailing asterisk (*), which allows for the trailing spaces:

```
WHERE       SKU LIKE '10?200*';
```

BY THE WAY

The SQL wildcard percent sign (%) and underscore (_) characters are specified in the SQL-92 standard. They are accepted by all DBMS products *except Microsoft Access*. So, why does Microsoft Access use the asterisk (*) character instead of the percent sign (%) and the question mark (?) instead of the underscore? These differences exist because Microsoft Access is, as we noted earlier, using the SQL-89 standard (which Microsoft calls ANSI-89 SQL). In that standard, the asterisk (*) and the question mark (?) are the correct wildcard characters. Switch a Microsoft Access database to to SQL-92 (which Microsoft calls ANSI-92 SQL) in Access Options dialog box, and the percent sign (%) and underscore (_) characters will work. For more information, see *http://office.microsoft.com/en-us/access-help/access-wildcard-character-reference-HA010076601.aspx#BMansi89*.

Combing the SQL WHERE Clause and the SQL ORDER BY Clause

If we want to sort the results generated by these enhanced SQL WHERE clauses, we simply combine the SQL ORDER BY clause with the WHERE clause. This is illustrated by the following SQL query:

```
/* *** SQL-Query-CH02-25 *** */
SELECT      *
FROM        ORDER_ITEM
WHERE       ExtendedPrice BETWEEN 100 AND 200
ORDER BY    OrderNumber DESC;
```

which will produce the following result:

	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	3000	101100	2	50.00	100.00
2	2000	101200	2	50.00	100.00
3	2000	101100	4	50.00	200.00
4	1000	202000	1	130.00	130.00

Performing Calculations in SQL Queries

It is possible to perform certain types of calculations in SQL query statements. One group of calculations involves the use of SQL built-in functions. Another group involves simple arithmetic operations on the columns in the SELECT statement. We will consider each, in turn.

Using SQL Built-in Functions

There are five **SQL built-in functions** for performing arithmetic on table columns: **SUM, AVG, MIN, MAX**, and **COUNT**. Some DBMS products extend these standard built-in functions by providing additional functions. Here we will focus only on the five standard SQL built-in functions.

Suppose we want to know the sum of OrderTotal for all of the orders in RETAIL_ORDER. We can obtain that sum by using the SQL built-in SUM function:

```
/* *** SQL-Query-CH02-26 *** */
SELECT      SUM(OrderTotal)
FROM        RETAIL_ORDER;
```

The result will be:

	(No column name)
1	1235.00

Recall that the result of an SQL statement is always a table. In this case, the table has one cell (the intersection of one row and one column that contains the sum of OrderTotal). But

because the OrderTotal sum is not a column in a table, the DBMS has no column name to provide. The preceding result was produced by Microsoft SQL Server 2012, and it names the column '(No column name)'. Other DBMS products take other, equivalent actions.

This result is ugly. We would prefer to have a meaningful column name, and SQL allows us to assign one using the **SQL AS keyword**. If we use the AS keyword in the query as follow:

```
/* *** SQL-Query-CH02-27 *** */
SELECT      SUM(OrderTotal) AS OrderSum
FROM        RETAIL_ORDER;
```

The result of this modified query will be:

	OrderSum
1	1235.00

This result has a much more meaningful column label. The name *OrderSum* is arbitrary—we are free to pick any name that we think would be meaningful to the user of the result. We could pick *OrderTotal_Total*, *OrderTotalSum*, or any other label that we think would be useful.

The utility of the built-in functions increases when you use them with an SQL WHERE clause. For example, we can write the SQL query:

```
/* *** SQL-Query-CH02-28 *** */
SELECT      SUM(ExtendedPrice) AS Order3000Sum
FROM        ORDER_ITEM
WHERE       OrderNumber=3000;
```

The result of this query is:

	Order3000Sum
1	450.00

The SQL built-in functions can be mixed and matched in a single statement. For example, we can create the following SQL statement:

```
/* *** SQL-Query-CH02-29 *** */
SELECT      SUM(ExtendedPrice) AS OrderItemSum,
            AVG(ExtendedPrice) AS OrderItemAvg,
            MIN(ExtendedPrice) AS OrderItemMin,
            MAX(ExtendedPrice) AS OrderItemMax
FROM        ORDER_ITEM;
```

The result of this query is:

	OrderItemSum	OrderItemAvg	OrderItemMin	OrderItemMax
1	1180.00	168.5714	50.00	300.00

The SQL built-in COUNT function sounds similar to the SUM function, but it produces very different results. The COUNT function *counts* the number of rows, whereas the SUM function *adds* the values in a column. For example, we can use the SQL built-in COUNT function to determine how many rows are in the ORDER_ITEM table:

```
/* *** SQL-Query-CH02-30 *** */
SELECT      COUNT(*) AS NumberOfRows
FROM        ORDER_ITEM;
```

The result of this query is:

	NumberOfRows
1	7

This result indicates that there are seven rows in the ORDER_ITEM table. Notice that we need to provide an asterisk (*) after the COUNT function when we want to count rows. COUNT is the only built-in function that requires a parameter, which can be the asterisk (as used in SQL-Query-CH02-30) or a column name (as used in SQL-Query-CH02-31 that follows). The COUNT function is also unique because it can be used on any type of data, but the SUM, AVG, MIN, and MAX functions can only be used with numeric data.

The COUNT function can produce some surprising results. For example, suppose you want to count the number of departments in the SKU_DATA table. If we use the following query:

```
/* *** SQL-Query-CH02-31 *** */
SELECT     COUNT(Department) AS DeptCount
FROM       SKU_DATA;
```

The result is:

	DeptCount
1	8

which is the number of rows in the SKU_DATA table, *not* the number of unique values of Department, as shown in Figure 2-5. If we want to count the unique values of Department, we need to use the SQL DISTINCT keyword, as follows:

```
/* *** SQL-Query-CH02-32 *** */
SELECT     COUNT(DISTINCT Department) AS DeptCount
FROM       SKU_DATA;
```

The result of this query is:

	DeptCount
1	3

Does Not Work with Microsoft Access ANSI-89 SQL

Microsoft Access does not support the DISTINCT keyword as part of the COUNT expression, so although the SQL command with COUNT(Department) will work, the SQL command with COUNT(DISTINCT Department) will fail.

Solution: Use an SQL subquery structure (discussed later in this chapter) with the DISTINCT keyword in the subquery itself. This SQL query works:

```
/* *** SQL-Query-CH02-32-Access *** */
SELECT     COUNT(*) AS DeptCount
FROM       (SELECT DISTINCT Department
            FROM SKU_DATA) AS DEPT;
```

Note that this query is a bit different from the other SQL queries using subqueries we show in this text because this subquery is in the FROM clause instead of (as you'll see) the WHERE clause. Basically, this subquery builds a new temporary table named DEPT containing only distinct Department values, and the query counts the number of those values.

You should be aware of two limitations to SQL built-in functions. First, except for grouping (defined later), you *cannot* combine a table column name with an SQL built-in function. For example, what happens if we run the following SQL query?

```
/* *** SQL-Query-CH02-33 *** */
SELECT     Department, COUNT(*)
FROM       SKU_DATA;
```

The result in Microsoft SQL Server 2012 is:

```
Msg 8120, Level 16, State 1, Line 1
Column 'SKU_DATA.Department' is invalid in the select list because it is not contained
in either an aggregate function or the GROUP BY clause.
```

This is the specific Microsoft SQL Server 2012 error message. However, you will receive an equivalent message from Microsoft Access, Oracle Database, DB2, or MySQL.

The second problem with the SQL built-in functions that you should understand is that you cannot use them in an SQL WHERE clause. Thus, you cannot use the following SQL statement:

```
/* *** SQL-Query-CH02-34 *** */
SELECT      *
FROM        RETAIL_ORDER
WHERE       OrderTotal > AVG(OrderTotal);
```

An attempt to use such a statement will also result in an error statement from the DBMS:

```
Msg 147, Level 15, State 1, Line 3
An aggregate may not appear in the WHERE clause unless it is in a subquery contained
in a HAVING clause or a select list, and the column being aggregated is an outer reference.
```

Again, this is the specific Microsoft SQL Server 2012 error message, but other DBMS products will give you an equivalent error message. In Chapter 7, you will learn how to obtain the desired result of the above query using a sequence of SQL views.

SQL Expressions in SQL SELECT Statements

It is possible to do basic arithmetic in SQL statements. For example, suppose we want to compute the values of extended price, perhaps because we want to verify the accuracy of the data in the ORDER_ITEM table. To compute the extended price, we can use the SQL expression Quantity * Price in the SQL query:

```
/* *** SQL-Query-CH02-35 *** */
SELECT      Quantity * Price AS EP
FROM        ORDER_ITEM;
```

The result is:

	EP
1	300.00
2	200.00
3	100.00
4	100.00
5	50.00
6	300.00
7	130.00

An **SQL expression** is basically a formula or set of values that determines the exact results of an SQL query. We can think of an SQL expression as anything that follows an actual or implied equal to (=) character (or any other relational operator, such as greater than [>], less than [<], and so on) or that follows certain SQL keywords, such as LIKE and BETWEEN. Thus, the SELECT clause in the preceding query includes the implied equal to (=) sign as EP = Quantity * Price. For another example, in the WHERE clause:

```
WHERE     Buyer IN ('Nancy Meyers', 'Cindy Lo', 'Jerry Martin');
```

the SQL expression consists of the enclosed set of three text values following the IN keyword.

Now that we know how to calculate the value of extended price, we can compare this computed value to the stored value of ExtendedPrice by using the SQL query:

```
/* *** SQL-Query-CH02-36 *** */
SELECT     Quantity * Price AS EP, ExtendedPrice
FROM       ORDER_ITEM;
```

The result of this statement now allows us to visually compare the two values to ensure that the stored data are correct:

	EP	ExtendedPrice
1	300.00	300.00
2	200.00	200.00
3	100.00	100.00
4	100.00	100.00
5	50.00	50.00
6	300.00	300.00
7	130.00	130.00

Another use for SQL expressions in SQL statements is to perform string manipulation. Suppose we want to combine (using the *concatenation* operator, which is the plus sign [+] in Microsoft SQL Server 2012) the Buyer and Department columns into a single column named Sponsor. To do this, we can use the SQL statement:

```
/* *** SQL-Query-CH02-37 *** */
SELECT     Buyer+' in '+Department AS Sponsor
FROM       SKU_DATA;
```

The result will include a column named Sponsor that contains the combined text values:

	Sponsor
1	Pete Hansen in Water Sports
2	Pete Hansen in Water Sports
3	Nancy Meyers in Water Sports
4	Nancy Meyers in Water Sports
5	Cindy Lo in Camping
6	Cindy Lo in Camping
7	Jerry Martin in Climbing
8	Jerry Martin in Climbing

BY THE WAY

The concatenation operator, like many SQL syntax elements, varies from one DBMS product to another. Oracle Database uses a double vertical bar [||] as the concatenation operator, and SQL –QUERY-CH02-37 is written for Oracle Database as:

```
/* *** SQL-Query-CH02-37-Oracle-Database *** */
SELECT     Buyer||' in '||Department AS Sponsor
FROM       SKU_DATA;
```

MySQL uses the concatenation string function CONCAT() as the concatenation operator with the elements to be concatenated separated by commas with the parentheses, and SQL-QUERY-CH02-37 is written for MySQL as:

```
/* *** SQL-Query-CH02-37-MySQL *** */
SELECT     CONCAT(Buyer,' in ',Department) AS Sponsor
FROM       SKU_DATA;
```

The result of SQL-Query-CH02-37 is ugly because of the extra spaces in each row. We can eliminate these extra spaces by using more advanced functions. The syntax and use of such functions vary from one DBMS to another, however, and a discussion of the features of each product will take us away from the point of this discussion. To learn more, search on *string functions* in the documentation for your specific DBMS product. Just to illustrate the possibilities, however, here is a Microsoft SQL Server 2012 statement using the RTRIM function that strips the tailing blanks off the right-hand side of Buyer and Department:

```
/* *** SQL-Query-CH02-38 *** */
SELECT     DISTINCT RTRIM(Buyer)+' in '+RTRIM(Department) AS Sponsor
FROM       SKU_DATA;
```

The result of this query is much more visually pleasing:

	Sponsor
1	Cindy Lo in Camping
2	Jerry Martin in Climbing
3	Nancy Meyers in Water Sports
4	Pete Hansen in Water Sports

Grouping in SQL SELECT Statements

In SQL queries, rows can be grouped according to common values using the **SQL GROUP BY clause**. For example, if you specify GROUP BY Department in a SELECT statement on the SKU_DATA table, the DBMS will first sort all rows by Department and then combine all of the rows having the same value into a group for that department. A grouping will be formed for each unique value of Department. For example, we can use the GROUP BY clause in the SQL query:

```
/* *** SQL-Query-CH02-39 *** */
SELECT     Department, COUNT(*) AS Dept_SKU_Count
FROM       SKU_DATA
GROUP BY   Department;
```

We get the result:

	Department	Dept_SKU_Count
1	Camping	2
2	Climbing	2
3	Water Sports	4

To obtain this result, the DBMS first sorts the rows according to Department and then counts the number of rows having the same value of Department.

Here is another example of an SQL query using GROUP BY:

```
/* *** SQL-Query-CH02-40 *** */
SELECT     SKU, AVG(ExtendedPrice) AS AvgEP
FROM       ORDER_ITEM
GROUP BY   SKU;
```

The result for this query is:

	SKU	AvgEP
1	100200	300.00
2	101100	150.00
3	101200	75.00
4	201000	300.00
5	202000	130.00

Here the rows have been sorted and grouped by SKU and the average ExtendedPrice for each group of SKU items has been calculated.

We can include more than one column in a GROUP BY expression. For example, the SQL statement:

```
/* *** SQL-Query-CH02-41 *** */
SELECT     Department, Buyer, COUNT(*) AS Dept_Buyer_SKU_Count
FROM       SKU_DATA
GROUP BY   Department, Buyer;
```

groups rows according to the value of Department first, then according to Buyer, and then counts the number of rows for each combination of Department and Buyer. The result is:

	Department	Buyer	Dept_Buyer_SKU_Count
1	Camping	Cindy Lo	2
2	Climbing	Jerry Martin	2
3	Water Sports	Nancy Meyers	2
4	Water Sports	Pete Hansen	2

When using the GROUP BY clause, *any and all* column names in the SELECT clause that are *not* used by or associated with an SQL built-in function *must* appear in the GROUP BY clause. In SQL-Query-CH02-42 below, the column name SKU is not used in the GROUP BY clause, and therefore the query produces an error:

```
/* *** SQL-Query-CH02-42 *** */
SELECT     SKU, Department, COUNT(*) AS Dept_SKU_Count
FROM       SKU_DATA
GROUP BY   Department;
```

The resulting error message is:

```
Msg 8120, Level 16, State 1, Line 1
Column 'SKU_DATA.SKU' is invalid in the select list because it is not contained
in either an aggregate function or the GROUP BY clause.
```

This is the specific Microsoft SQL Server 2012 error message, but other DBMS products will give you an equivalent error message. Statements like this one are invalid because there are many values of SKU for each Department group. The DBMS has no place to put those multiple values in the result. If you do not understand the problem, try to process this statement by hand. It cannot be done.

Of course, the SQL WHERE and ORDER BY clauses can also be used with SELECT statements, as shown in the following query:

```
/* *** SQL-Query-CH02-43 *** */
SELECT     Department, COUNT(*) AS Dept_SKU_Count
FROM       SKU_DATA
WHERE      SKU <> 302000
GROUP BY   Department
ORDER BY   Dept_SKU_Count;
```

The result is:

	Department	Dept_SKU_Count
1	Climbing	1
2	Camping	2
3	Water Sports	4

Notice that one of the rows of the Climbing department has been removed from the count because it did not meet the WHERE clause condition. Without the ORDER BY clause, the rows would be presented in arbitrary order of Department. With it, the order is as shown. In general, to be safe, always place the WHERE clause before the GROUP BY clause. Some DBMS products do not require that placement, but others do.

Does Not Work with Microsoft Access ANSI-89 SQL

Microsoft Access does not properly recognize the alias Dept_SKU_Count in the ORDER BY clause and creates a parameter query that requests an input value of as yet nonexistent Dept_SKU_Count! However, it doesn't matter whether you enter parameter values or not—click the OK button and the query will run. The results will be basically correct, but they will not be sorted correctly.

Solution: Use the Microsoft Access QBE GUI to modify the query structure. The correct QBE structure is shown in Figure 2-22. The resulting Microsoft Access ANSI-89 SQL is:

```
/* *** SQL-Query-CH02-43-Access-A *** */
SELECT      SKU_DATA.Department, Count(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       (((SKU_DATA.SKU)<>302000))
GROUP BY    SKU_DATA.Department
ORDER BY    Count(*);
```

which can be edited down to:

```
/* *** SQL-Query-CH02-43-Access-B *** */
SELECT      Department, Count(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       SKU<>302000
GROUP BY    Department
ORDER BY    Count(*);
```

Edit the query in the QBE GUI interface so that it appears as shown here

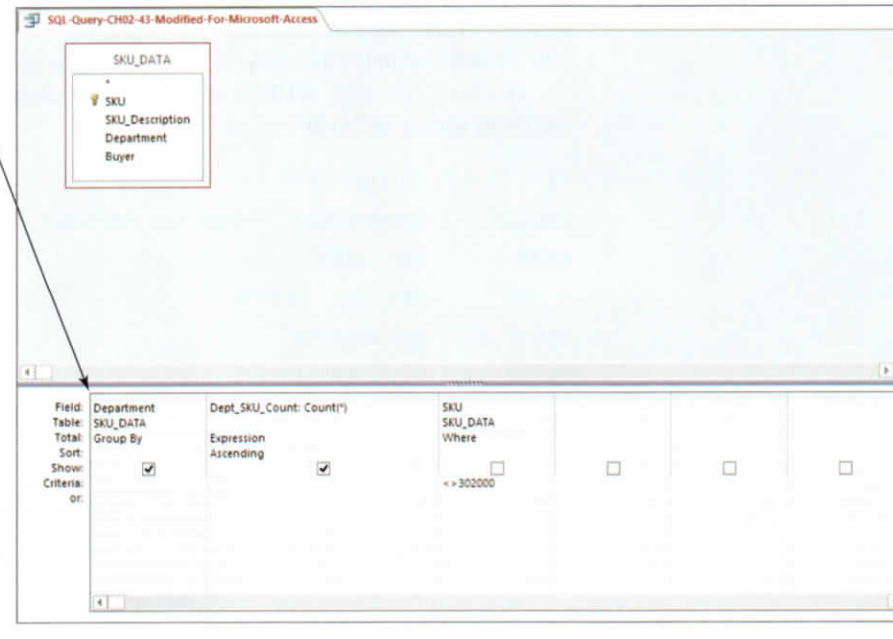


Figure 2-22
Editing the SQL Query in the Microsoft Access 2013 QBE GUI Interface

SQL provides one more GROUP BY clause feature that extends its functionality even further. The **SQL HAVING clause** restricts the groups that are presented in the result. We can restrict the previous query to display only groups having more than one row by using the SQL query:

```
/* *** SQL-Query-CH02-44 *** */
SELECT      Department, COUNT(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       SKU <> 302000
GROUP BY    Department
HAVING      COUNT (*) > 1
ORDER BY    Dept_SKU_Count;
```

The result of this modified query is:

	Department	Dept_SKU_Count
1	Camping	2
2	Water Sports	4

Comparing this result with the previous one, the row for Climbing (which has a count of 1) has been eliminated.

Does Not Work with Microsoft Access ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason as the previous query.

Solution: See the solution described in the previous "Does Not Work with Microsoft Access ANSI-89 SQL" box. The correct Microsoft Access ANSI-89 SQL for this query is:

```
/* *** SQL-Query-CH02-44-Access *** */
SELECT      Department, Count(*) AS Dept_SKU_Count
FROM        SKU_DATA
WHERE       SKU<>302000
GROUP BY    Department
HAVING      Count(*)>1
ORDER BY    Count(*);
```

SQL built-in functions can be used in the HAVING clause. For example, the following is a valid SQL query:

```
/* *** SQL-Query-CH02-45 *** */
SELECT     COUNT(*) AS SKU_Count, SUM(Price) AS TotalRevenue, SKU
FROM       ORDER_ITEM
GROUP BY   SKU
HAVING     SUM(Price)=100;
```

The results for this query are:

	SKU_Count	TotalRevenue	SKU
1	2	100.00	101100
2	2	100.00	101200

Be aware that there is an ambiguity in statements that include both WHERE and HAVING clauses. The results vary depending on whether the WHERE condition is applied before or after the HAVING. To eliminate this ambiguity, the WHERE clause is *always* applied before the HAVING clause.

Looking for Patterns in NASDAQ Trading

Before we continue our discussion of SQL, consider an example problem that will illustrate the power of the SQL just described.

Suppose that a friend tells you that she suspects the stock market tends to go up on certain days of the week and down on others. She asks you to investigate past trading data to determine if this is true. Specifically, she wants to trade an index fund called the NASDAQ 100, which is a stock fund of the 100 top companies traded on the NASDAQ stock exchange. She gives you a dataset with 20 years (1985–2004) of NASDAQ 100 trading data for analysis. Assume she gives you the data in the form of a table named NDX containing 4611 rows of data for use with a relational database (this dataset is available on the text's Web site at *www.pearsonhighered.com/kroenke*).

Investigating the Characteristics of the Data

Suppose you first decide to investigate the general characteristics of the data. You begin by seeing what columns are present in the table by issuing the SQL query:

```
/* *** SQL-Query-NDX-CH02-01 *** */
SELECT     *
FROM       NDX;
```

The first five rows of that query are as follows:

	TClose	PriorClose	ChangeClose	Volume	TMonth	TDayOfMonth	TYear	TDayOfWeek	TQuarter
1	1520.46	1530.65	-10.1900000000001	24827600	January	9	2004	Friday	1
2	1530.65	1514.26	16.3900000000001	26839500	January	8	2004	Thursday	1
3	1514.26	1501.26	13	22942800	January	7	2004	Wednesday	1
4	1501.26	1496.58	4.68000000000006	22732200	January	6	2004	Tuesday	1
5	1496.58	1463.57	33.01	23629100	January	5	2004	Monday	1

BY THE WAY

To control how many rows an SQL query displays, use the **SQL TOP {NumberOfRows} property**. To show the top five rows in SQL-Query-NDX-CH02-02, modify it as:

```
/* *** SQL-Query-NDX-CH02-01A *** */
SELECT     TOP 5 *
FROM       NDX;
```

Assume that you learn that the first column has the value of the fund at the close of a trading day, the second column has the value of the fund at the close of the prior trading day,

and the third row has the difference between the current day's close and the prior day's close. Volume is the number of shares traded, and the rest of the data concern the trading date.

Next, you decide to investigate the change of the stock price by issuing the SQL query:

```
/* *** SQL-Query-NDX-CH02-02 *** */
SELECT     AVG(ChangeClose) AS AverageChange,
           MAX(ChangeClose) AS MaxGain,
           MIN(ChangeClose) AS MaxLoss
FROM       NDX;
```

The result of this query is:

	AverageChange	MaxGain	MaxLoss
1	0.281167028199584	399.6	-401.03

DBMS products have many functions for formatting query results to reduce the number of decimal points displayed, to add currency characters such as $ or £, or to make other formatting changes. However, these functions are DBMS-dependent. Search the documentation of your DBMS for the term *formatting results* to learn more about such functions.

Just out of curiosity, you decide to determine which days had the maximum and minimum change. To avoid having to key in the long string of decimal places that would be required to make an equal comparison, you use a greater than and less than comparison with values that are close:

```
/* *** SQL-Query-NDX-CH02-03 *** */
SELECT     ChangeClose, TMonth, TDayOfMonth, TYear
FROM       NDX
WHERE      ChangeClose > 398
    OR     ChangeClose < -400;
```

The result is:

	ChangeClose	TMonth	TDayOfMonth	TYear
1	-401.03	January	3	1994
2	399.6	January	3	2001

This result is surprising! Is there some reason that both the greatest loss and the greatest gain both occurred on January 3? You begin to wonder if your friend might have a promising idea.

Searching for Patterns in Trading by Day of Week

You want to determine if there is a difference in the average trade by day of week. Accordingly, you create the SQL query:

```
/* *** SQL-Query-NDX-CH02-04 *** */
SELECT     TDayOfWeek, AVG(ChangeClose) AS AvgChange
FROM       NDX
GROUP BY   TDayOfWeek;
```

The result is:

	TDayOfWeek	AvgChange
1	Wednesday	0.777940552017005
2	Monday	-1.03577929465299
3	Friday	0.146021739130452
4	Thursday	2.17412972972975
5	Tuesday	-0.711440677966085

Indeed, there does seem to be a difference according to the day of the week. The NASDAQ 100 appears to go down on Monday and Tuesday and then go up on the other three days of the week. Thursday, in particular, seems to be a good day to trade long.

But, you begin to wonder, is this pattern true for each year? To answer that question, you use the query:

```
/* *** SQL-Query-NDX-CH02-05 *** */
SELECT     TDayOfWeek, TYear, AVG(ChangeClose) AS AvgChange
FROM       NDX
GROUP BY   TDayOfWeek, TYear
ORDER BY   TDayOfWeek, TYear DESC;
```

Because there are 20 years of data, this query results in 100 rows, of which the first 12 are shown in the following results:

	TDayOfWeek	TYear	AvgChange
1	Friday	2004	-7.2700000000001
2	Friday	2003	-2.48499999999996
3	Friday	2002	-2.19419999999997
4	Friday	2001	-19.5944
5	Friday	2000	8.8980392156863
6	Friday	1999	13.9656000000001
7	Friday	1998	5.2640816326531
8	Friday	1997	-0.194799999999989
9	Friday	1996	0.819019607843153
10	Friday	1995	0.691372549019617
11	Friday	1994	0.123725490196082
12	Friday	1993	-0.899399999999989

To simplify your analysis, you decide to restrict the number of rows to the most recent 5 years (2000–2004):

```
/* *** SQL-Query-NDX-CH02-06 *** */
SELECT     TDayOfWeek, TYear, AVG(ChangeClose) AS AvgChange
FROM       NDX
WHERE      TYear > '1999'
GROUP BY   TDayOfWeek, TYear
ORDER BY   TDayOfWeek, TYear DESC;
```

Partial results from this query are as follows:

	TDayOfWeek	TYear	AvgChange
1	Friday	2004	-7.2700000000001
2	Friday	2003	-2.48499999999996
3	Friday	2002	-2.19419999999997
4	Friday	2001	-19.5944
5	Friday	2000	8.8980392156863
6	Monday	2004	33.01
7	Monday	2003	3.77416666666668
8	Monday	2002	-2.60229166666664
9	Monday	2001	-3.7527083333333
10	Monday	2000	-19.899574468085
11	Thursday	2004	16.3900000000001
12	Thursday	2003	5.70700000000002
13	Thursday	2002	-3.77979999999998
14	Thursday	2001	9.31440000000003
15	Thursday	2000	24.766274509804
16	Tuesday	2004	4.68000000000006
17	Tuesday	2003	4.41307692307694
18	Tuesday	2002	-7.85882352941176
19	Tuesday	2001	-8.88459999999997
20	Tuesday	2000	-3.50627450980385
21	Wednesday	2004	13
22	Wednesday	2003	-1.69596153846152
23	Wednesday	2002	4.54372549019611
24	Wednesday	2001	7.47420000000005
25	Wednesday	2000	-37.8636538461538

Alas, it does not appear that day of week is a very good predictor of gain or loss. At least, not for this fund over this period of time. We could continue this discussion to further analyze this data, but by now you should understand how useful SQL can be for analyzing and processing a table. Suggested additional NDX analysis exercises are included in the SQL problems at the end of this chapter.

Querying Two or More Tables with SQL

So far in this chapter we've worked with only one table. Now we will conclude by describing SQL statements for querying two or more tables.

Suppose that you want to know the revenue generated by SKUs managed by the Water Sports department. We can compute revenue as the sum of ExtendedPrice, but we have a problem. ExtendedPrice is stored in the ORDER_ITEM table, and Department is stored in the SKU_DATA table. We need to process data in two tables, and all of the SQL presented so far operates on a single table at a time.

SQL provides two different techniques for querying data from multiple tables: subqueries and joins. Although both work with multiple tables, they are used for slightly different purposes, as you will learn.

Querying Multiple Tables with Subqueries

How can we obtain the sum of ExtendedPrice for items managed by the Water Sports department? If we somehow knew the SKU values for those items, we could use a WHERE clause with the IN keyword.

For the data in Figure 2-5, the SKU values for items in Water Sports are 100100, 100200, 101100, and 101200. Knowing those values, we can obtain the sum of their ExtendedPrice with the following SQL query:

```
/* *** SQL-Query-CH02-46 *** */
SELECT     SUM(ExtendedPrice) AS Revenue
FROM       ORDER_ITEM
WHERE      SKU IN (100100, 100200, 101100, 101200);
```

The result is:

	Revenue
1	750.00

But, in general, we do not know the necessary SKU values ahead of time. However, we do have a way to obtain them from an SQL query on the data in the SKU_DATA table. To obtain the SKU values for the Water Sports department, we use the SQL statement:

```
/* *** SQL-Query-CH02-47 *** */
SELECT     SKU
FROM       SKU_DATA
WHERE      Department='Water Sports'
```

The result of this SQL statement is:

	SKU
1	100100
2	100200
3	101100
4	101200

which is, indeed, the desired list of SKU values.

Now we need only combine the last two SQL statements to obtain the result we want. We replace the list of values in the WHERE clause of the first SQL query with the second SQL statement as follows:

```
/* *** SQL-Query-CH02-48 *** */
SELECT     SUM(ExtendedPrice) AS Revenue
FROM       ORDER_ITEM
WHERE      SKU IN
           (SELECT   SKU
            FROM     SKU_DATA
            WHERE    Department='Water Sports');
```

The result of the query is:

	Revenue
1	750.00

which is the same result we obtained before when we knew which values of SKU to use.

In the preceding SQL query, the second SELECT statement, the one enclosed in parentheses, is called a **subquery**. We can use multiple subqueries to process three or even more tables. For example, suppose we want to know the names of the buyers who manage any product purchased in January 2013. First, note that Buyer data is stored in the SKU_DATA table and OrderMonth and OrderYear data are stored in the RETAIL_ORDER table.

Now, we can use an SQL query with two subqueries to obtain the desired data as follows:

```
/* *** SQL-Query-CH02-49 *** */
SELECT     Buyer
FROM       SKU_DATA
WHERE      SKU IN
           (SELECT    SKU
            FROM      ORDER_ITEM
            WHERE     OrderNumber  IN
                      (SELECT      OrderNumber
                       FROM        RETAIL_ORDER
                       WHERE       OrderMonth='January'
                           AND     OrderYear=2013));
```

The result of this statement is:

	Buyer
1	Pete Hansen
2	Nancy Meyers
3	Nancy Meyers

To understand this statement, work from the bottom up. The bottom SELECT statement obtains the list of OrderNumbers of orders sold in January 2013. The middle SELECT statement obtains the SKU values for items sold in orders in January 2013. Finally, the top-level SELECT query obtains Buyer for all of the SKUs found in the middle SELECT statement.

Any parts of the SQL language that you learned earlier in this chapter can be applied to a table generated by a subquery, regardless of how complicated the SQL looks. For example, we can apply the DISTINCT keyword on the results to eliminate duplicate rows. Or we can apply the GROUP BY and ORDER BY clauses as follows:

```
/* *** SQL-Query-CH02-50 *** */
SELECT     Buyer, COUNT(*) AS NumberSold
FROM       SKU_DATA
WHERE      SKU IN
           (SELECT    SKU
            FROM      ORDER_ITEM
            WHERE     OrderNumber  IN
                      (SELECT      OrderNumber
                       FROM        RETAIL_ORDER
                       WHERE       OrderMonth='January'
                           AND     OrderYear=2013))
GROUP BY   Buyer
ORDER BY   NumberSold DESC;
```

The result is:

	Buyer	NumberSold
1	Nancy Meyers	2
2	Pete Hansen	1

Does Not Work with Microsoft Access ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason previously described on page 72.

Solution: See the solution described in the "Does Not Work with Microsoft Access ANSI-89 SQL" box on page 72. The correct Microsoft Access ANSI-89 SQL statement for this query is:

```
/* *** SQL-Query-CH02-50-Access *** */
SELECT   Buyer, Count(*) AS NumberSold
FROM     SKU_DATA
WHERE    SKU IN
         (SELECT    SKU
          FROM      ORDER_ITEM
          WHERE     OrderNumber  IN
                    (SELECT      OrderNumber
                     FROM        RETAIL_ORDER
                     WHERE       OrderMonth='January'
                         AND     OrderYear=2011))
GROUP    BY Buyer
ORDER    BY Count(*) DESC;
```

Querying Multiple Tables with Joins

Subqueries are very powerful, but they do have a serious limitation. The selected data can only come from the top-level table. We cannot use a subquery to obtain data that arise from more than one table. To do so, we must use a join instead.

The **SQL join operator** is used to combine two or more tables by concatenating (sticking together) the rows of one table with the rows of another table. Consider how we might combine the data in the RETAIL_ORDER and ORDER_ITEM tables. We can concatenate the rows of one table with the rows of the second table with the following SQL statement, where we simply list the names of the tables we want to combine:

```
/* *** SQL-Query-CH02-51 *** */
SELECT     *
FROM       RETAIL_ORDER, ORDER_ITEM;
```

This statement will just stick every row of one table together with every row of the second table. For the data in Figure 2-5, the result is:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2012	445.00	3000	100200	1	300.00	300.00
2	1000	10	98110	December	2012	445.00	2000	101100	4	50.00	200.00
3	1000	10	98110	December	2012	445.00	3000	101100	2	50.00	100.00
4	1000	10	98110	December	2012	445.00	2000	101200	2	50.00	100.00
5	1000	10	98110	December	2012	445.00	3000	101200	1	50.00	50.00
6	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00
7	1000	10	98110	December	2012	445.00	1000	202000	1	130.00	130.00
8	2000	20	02335	December	2012	310.00	3000	100200	1	300.00	300.00
9	2000	20	02335	December	2012	310.00	2000	101100	4	50.00	200.00
10	2000	20	02335	December	2012	310.00	3000	101100	2	50.00	100.00
11	2000	20	02335	December	2012	310.00	2000	101200	2	50.00	100.00
12	2000	20	02335	December	2012	310.00	3000	101200	1	50.00	50.00
13	2000	20	02335	December	2012	310.00	1000	201000	1	300.00	300.00
14	2000	20	02335	December	2012	310.00	1000	202000	1	130.00	130.00
15	3000	10	98110	January	2013	480.00	3000	100200	1	300.00	300.00
16	3000	10	98110	January	2013	480.00	2000	101100	4	50.00	200.00
17	3000	10	98110	January	2013	480.00	3000	101100	2	50.00	100.00
18	3000	10	98110	January	2013	480.00	2000	101200	2	50.00	100.00
19	3000	10	98110	January	2013	480.00	3000	101200	1	50.00	50.00
20	3000	10	98110	January	2013	480.00	1000	201000	1	300.00	300.00
21	3000	10	98110	January	2013	480.00	1000	202000	1	130.00	130.00

Because there are 3 rows of retail order and 7 rows of order items, there are 3 times 7, or 21, rows in this table. Notice that the retail order with OrderNumber 1000 has been combined with all seven of the rows in ORDER_ITEM, the retail order with OrderNumber 2000 has been combined with all seven of the same rows, and, finally, the retail order with OrderNumber 3000 has again been combined with all seven rows.

This is illogical—what we need to do is to select only those rows for which the OrderNumber of RETAIL_ORDER matches the OrderNumber in ORDER_ITEM. This is easy to do; we simply add an SQL WHERE clause to the query:

```
/* *** SQL-Query-CH02-52 *** */
SELECT      *
FROM        RETAIL_ORDER, ORDER_ITEM
WHERE       RETAIL_ORDER.OrderNumber=ORDER_ITEM.OrderNumber;
```

The result is:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	3000	10	98110	January	2013	480.00	3000	100200	1	300.00	300.00
2	2000	20	02335	December	2012	310.00	2000	101100	4	50.00	200.00
3	3000	10	98110	January	2013	480.00	3000	101100	2	50.00	100.00
4	2000	20	02335	December	2012	310.00	2000	101200	2	50.00	100.00
5	3000	10	98110	January	2013	480.00	3000	101200	1	50.00	50.00
6	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00
7	1000	10	98110	December	2012	445.00	1000	202000	1	130.00	130.00

This is technically correct, but it will be easier to read if we sort the results using an ORDER BY clause:

```
/* *** SQL-Query-CH02-53 *** */
SELECT      *
FROM        RETAIL_ORDER, ORDER_ITEM
WHERE       RETAIL_ORDER.OrderNumber=ORDER_ITEM.OrderNumber
ORDER BY    RETAIL_ORDER.OrderNumber, ORDER_ITEM.SKU;
```

The result is:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00
2	1000	10	98110	December	2012	445.00	1000	202000	1	130.00	130.00
3	2000	20	02335	December	2012	310.00	2000	101100	4	50.00	200.00
4	2000	20	02335	December	2012	310.00	2000	101200	2	50.00	100.00
5	3000	10	98110	January	2013	480.00	3000	100200	1	300.00	300.00
6	3000	10	98110	January	2013	480.00	3000	101100	2	50.00	100.00
7	3000	10	98110	January	2013	480.00	3000	101200	1	50.00	50.00

If you compare this result with the data in Figure 2-5, you will see that only the appropriate order items are associated with each retail order. You also can tell that this has been done by noticing that, in each row, the value of OrderNumber from RETAIL_ORDER (the first column) equals the value of OrderNumber from ORDER_ITEM (the seventh column). This was not true for our first result.

You can think of the join operation working as follows. Start with the first row in RETAIL_ORDER. Using the value of OrderNumber in this first row (1000 for the data in Figure 2-5), examine the rows in ORDER_ITEM. When you find a row in ORDER_ITEM where OrderNumber

is also equal to 1000, join all the columns of the first row of RETAIL_ORDER with the columns from the row you just found in ORDER_ITEM.

For the data in Figure 2-5, the first row of ORDER_ITEM has OrderNumber equal to 1000, so you join the first row of RETAIL_ORDER with the columns from the first row in ORDER_ITEM to form the first row of the join. The result is:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00

Now, still using the OrderNumber value of 1000, look for a second row in ORDER_ITEM that has OrderNumber equal to 1000. For our data, the second row of ORDER_ITEM has such a value. So, join FirstName and LastName from the first row of RETAIL_ORDER to the second row of ORDER_ITEM to obtain the second row of the join, as follows:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00
2	1000	10	98110	December	2012	445.00	1000	202000	1	130.00	130.00

Continue in this way, looking for matches for the OrderNumber value of 1000. At this point, no more OrderNumber values of 1000 appear in the sample data, so now you move to the second row of RETAIL_ORDER, obtain the new value of OrderNumber (2000), and begin searching for matches for it in the rows of ORDER_ITEM. In this case, the third row has such a match, so you combine those rows with the previous result to obtain the new result:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00
2	1000	10	98110	December	2012	445.00	1000	202000	1	130.00	130.00
3	2000	20	02335	December	2012	310.00	2000	101100	4	50.00	200.00

You continue until all rows of RETAIL_ORDER have been examined. The final result is:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00
2	1000	10	98110	December	2012	445.00	1000	202000	1	130.00	130.00
3	2000	20	02335	December	2012	310.00	2000	101100	4	50.00	200.00
4	2000	20	02335	December	2012	310.00	2000	101200	2	50.00	100.00
5	3000	10	98110	January	2013	480.00	3000	100200	1	300.00	300.00
6	3000	10	98110	January	2013	480.00	3000	101100	2	50.00	100.00
7	3000	10	98110	January	2013	480.00	3000	101200	1	50.00	50.00

Actually, that is the theoretical result. But remember that row order in an SQL query can be arbitrary, as is shown in the results to SQL-Query-CH02-52 above. To ensure that you get the above result, you need to add an ORDER BY clause to the query, as shown in SQL-Query-CH02-53 above.

You may have noticed that we introduced a new variation in SQL statement syntax in the previous two queries, where the terms RETAIL_ORDER.OrderNumber, ORDER_ITEM.OrderNumber, and ORDER_ITEM.SKU were used. The new syntax is simply **TableName.ColumnName**, and it is used to specify exactly which table each column is linked to. RETAIL_ORDER.OrderNumber simply means the OrderNumber from the RETAIL_ORDER table. Similarly, ORDER_ITEM.OrderNumber refers to the OrderNumber in the ORDER_ITEM

table, and ORDER_ITEM.SKU refers to the SKU column in the ORDER_ITEM table. You can always qualify a column name with the name of its table like this. We have not done so previously because we were working with only one table, but the SQL statements shown previously would have worked just as well with syntax like SKU_DATA.Buyer rather than just Buyer or ORDER_ITEM.Price instead of Price.

The table that is formed by concatenating two tables is called a **join**. The process of creating such a table is called **joining the two tables**, and the associated operation is called a **join operation**. When the tables are joined using an equal condition (like the one on OrderNumber), this join is called an **equijoin**. When people say *join*, 99.99999 percent of the time they mean an *equijoin*. This type of join is also referred to as an **inner join**.

We can use a join to obtain data from two or more tables. For example, using the data in Figure 2-5, suppose we want to show the name of the Buyer and the ExtendedPrice of the sales of all items managed by that Buyer. The following SQL query will obtain that result:

```
/* *** SQL-Query-CH02-54 *** */
SELECT     Buyer, ExtendedPrice
FROM       SKU_DATA, ORDER_ITEM
WHERE      SKU_DATA.SKU=ORDER_ITEM.SKU;
```

The result is:

	Buyer	ExtendedPrice
1	Pete Hansen	300.00
2	Nancy Meyers	200.00
3	Nancy Meyers	100.00
4	Nancy Meyers	100.00
5	Nancy Meyers	50.00
6	Cindy Lo	300.00
7	Cindy Lo	130.00

Again, the result of every SQL statement is just a single table, so we can apply any of the SQL syntax you learned for a single table to this result. For example, we can use the GROUP BY and ORDER BY clauses to obtain the total revenue associated with each buyer, as shown in the following SQL query:

```
/* *** SQL-Query-CH02-55 *** */
SELECT     Buyer, SUM(ExtendedPrice) AS BuyerRevenue
FROM       SKU_DATA, ORDER_ITEM
WHERE      SKU_DATA.SKU=ORDER_ITEM.SKU
GROUP BY   Buyer
ORDER BY   BuyerRevenue DESC;
```

The result is:

	Buyer	BuyerRevenue
1	Nancy Meyers	450.00
2	Cindy Lo	430.00
3	Pete Hansen	300.00

Does Not Work with Microsoft Access ANSI-89 SQL

This query fails in Microsoft Access ANSI-89 SQL for the same reason previously described on page 72.

Solution: See the solution described in the "Does Not Work with Microsoft Access ANSI-89 SQL" box on page 72. The correct Microsoft Access ANSI-89 SQL statement for this query is:

```
/* *** SQL-Query-CH02-55-Access *** */
SELECT      Buyer, Sum(ORDER_ITEM.ExtendedPrice) AS BuyerRevenue
FROM        SKU_DATA, ORDER_ITEM
WHERE       SKU_DATA.SKU=ORDER_ITEM.SKU
GROUP BY    Buyer
ORDER BY    Sum(ExtendedPrice) DESC;
```

We can extend this syntax to join three or more tables. For example, suppose we want to obtain the Buyer and the ExtendedPrice and OrderMonth for all purchases of items managed by each buyer. To retrieve that data, we need to join all three tables together, as shown in this SQL query:

```
/* *** SQL-Query-CH02-56 *** */
SELECT     Buyer, ExtendedPrice, OrderMonth
FROM       SKU_DATA, ORDER_ITEM, RETAIL_ORDER
WHERE      SKU_DATA.SKU=ORDER_ITEM.SKU
    AND    ORDER_ITEM.OrderNumber=RETAIL_ORDER.OrderNumber;
```

The result is:

	Buyer	ExtendedPrice	OrderMonth
1	Pete Hansen	300.00	January
2	Nancy Meyers	200.00	December
3	Nancy Meyers	100.00	January
4	Nancy Meyers	100.00	December
5	Nancy Meyers	50.00	January
6	Cindy Lo	300.00	December
7	Cindy Lo	130.00	December

We can improve this result by sorting with the ORDER BY clause and grouping by Buyer with the GROUP BY clause:

```
/* *** SQL-Query-CH02-57 *** */
SELECT     Buyer, OrderMonth, SUM(ExtendedPrice) AS BuyerRevenue
FROM       SKU_DATA, ORDER_ITEM, RETAIL_ORDER
WHERE      SKU_DATA.SKU=ORDER_ITEM.SKU
    AND    ORDER_ITEM.OrderNumber=RETAIL_ORDER.OrderNumber
GROUP BY   Buyer, OrderMonth
ORDER BY   Buyer, OrderMonth DESC;
```

The result is:

	Buyer	OrderMonth	BuyerRevenue
1	Cindy Lo	December	430.00
2	Nancy Meyers	December	300.00
3	Nancy Meyers	January	150.00
4	Pete Hansen	January	300.00

Comparing Subqueries and Joins

Subqueries and joins both process multiple tables, but they differ slightly. As mentioned earlier, a subquery can only be used to retrieve data from the top table. A join can be used to obtain data from any number of tables. Thus, a join can do everything a subquery can do, and more. So why learn subqueries? For one, if you just need data from a single table, you might use a subquery because it is easier to write and understand. This is especially true when processing multiple tables.

In Chapter 8, however, you will learn about a type of subquery called a **correlated subquery**. A correlated subquery can do work that is not possible with joins. Thus, it is important for you to learn about both joins and subqueries, even though right now it appears that joins are uniformly superior. If you're curious, ambitious, and courageous, jump ahead and read the discussion of correlated subqueries on pages 368–370.

The SQL JOIN ON Syntax

So far, we have learned to code SQL joins using the following syntax:

```
/* *** SQL-Query-CH02-53 *** */
SELECT      *
FROM        RETAIL_ORDER, ORDER_ITEM
WHERE       RETAIL_ORDER.OrderNumber=ORDER_ITEM.OrderNumber
ORDER BY    RETAIL_ORDER.OrderNumber, ORDER_ITEM.SKU;
```

However, there is another way to code this join. In this second case, we use the **SQL JOIN ON syntax**:

```
/* *** SQL-Query-CH02-58 *** */
SELECT      *
FROM        RETAIL_ORDER JOIN ORDER_ITEM
    ON      RETAIL_ORDER.OrderNumber=ORDER_ITEM.OrderNumber
ORDER BY    RETAIL_ORDER.OrderNumber, ORDER_ITEM.SKU;
```

The result is:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00
2	1000	10	98110	December	2012	445.00	1000	202000	1	130.00	130.00
3	2000	20	02335	December	2012	310.00	2000	101100	4	50.00	200.00
4	2000	20	02335	December	2012	310.00	2000	101200	2	50.00	100.00
5	3000	10	98110	January	2013	480.00	3000	100200	1	300.00	300.00
6	3000	10	98110	January	2013	480.00	3000	101100	2	50.00	100.00
7	3000	10	98110	January	2013	480.00	3000	101200	1	50.00	50.00

These two join syntaxes are equivalent, and it is a matter of personal preference which one you use. Some people think that the SQL JOIN ON syntax is easier to understand than the first. Note that when using the SQL JOIN ON syntax:

- The **SQL JOIN keyword** is placed between the table names in the SQL FROM clause, where it replaces the comma that previously separated the two table names, and
- The **SQL ON keyword** now leads into an SQL ON clause, which includes the statement of matching key values that was previously in an SQL WHERE clause.

Note that the SQL ON clause does *not* replace the SQL WHERE clause, which can still be used to determine which rows will be displayed. For example, we can use the SQL WHERE clause to limit the records shown to those for the OrderYear of 2012:

```
/* *** SQL-Query-CH02-59 *** */
SELECT      *
FROM        RETAIL_ORDER JOIN ORDER_ITEM
    ON      RETAIL_ORDER.OrderNumber=ORDER_ITEM.OrderNumber
WHERE       OrderYear = '2012'
ORDER BY    RETAIL_ORDER.OrderNumber, ORDER_ITEM.SKU;
```

The result is:

	OrderNumber	StoreNumber	StoreZIP	OrderMonth	OrderYear	OrderTotal	OrderNumber	SKU	Quantity	Price	ExtendedPrice
1	1000	10	98110	December	2012	445.00	1000	201000	1	300.00	300.00
2	1000	10	98110	December	2012	445.00	1000	202000	1	130.00	130.00
3	2000	20	02335	December	2012	310.00	2000	101100	4	50.00	200.00
4	2000	20	02335	December	2012	310.00	2000	101200	2	50.00	100.00

You can use the SQL JOIN ON syntax as an alternate format for joins of three or more tables as well. If, for example, you want to obtain a list of the order data, order line data and SKU data, you can use the following SQL statement:

```
/* *** SQL-Query-CH02-60 *** */
SELECT      RETAIL_ORDER.OrderNumber, StoreNumber, OrderYear,
            ORDER_ITEM.SKU, SKU_Description, Department
FROM        RETAIL_ORDER JOIN ORDER_ITEM
    ON      RETAIL_ORDER.OrderNumber=ORDER_ITEM.OrderNumber
            JOIN SKU_DATA
                ON  ORDER_ITEM.SKU=SKU_DATA.SKU
WHERE       OrderYear = '2012'
ORDER BY    RETAIL_ORDER.OrderNumber, ORDER_ITEM.SKU;
```

The result is:

	OrderNumber	StoreNumber	OrderYear	SKU	SKU_Description	Department
1	1000	10	2012	201000	Half-dome Tent	Camping
2	1000	10	2012	202000	Half-dome Tent Vestibule	Camping
3	2000	20	2012	101100	Dive Mask, Small Clear	Water Sports
4	2000	20	2012	101200	Dive Mask, Med Clear	Water Sports

You can make that statement even simpler by using the SQL AS keyword to create table aliases as well as for naming output columns:

```
/* *** SQL-Query-CH02-61 *** */
SELECT      RO.OrderNumber, StoreNumber, OrderYear,
            OI.SKU, SKU_Description, Department
FROM        RETAIL_ORDER AS RO JOIN ORDER_ITEM AS OI
    ON      RO.OrderNumber=OI.OrderNumber
            JOIN SKU_DATA AS SD
                ON  OI.SKU=SD.SKU
WHERE       OrderYear = '2012'
ORDER BY    RO.OrderNumber, OI.SKU;
```

When a query produces a result table with many rows, we may want to limit the number of rows that we see. We can do this using the **SQL TOP {NumberOfRows} property**, which produces our final SQL query statement:

```
/* *** SQL-Query-CH02-62 *** */
SELECT     TOP 3 RO.OrderNumber, StoreNumber, OrderYear,
           OI.SKU, SKU_Description, Department
FROM       RETAIL_ORDER AS RO JOIN ORDER_ITEM AS OI
    ON     RO.OrderNumber=OI.OrderNumber
           JOIN SKU_DATA AS SD
                ON  OI.SKU=SD.SKU
WHERE      OrderYear = '2012'
ORDER BY   RO.OrderNumber, OI.SKU;
```

The result of this statement is:

	OrderNumber	StoreNumber	OrderYear	SKU	SKU_Description	Department
1	1000	10	2012	201000	Half-dome Tent	Camping
2	1000	10	2012	202000	Half-dome Tent Vestibule	Camping
3	2000	20	2012	101100	Dive Mask, Small Clear	Water Sports

Outer Joins

Suppose that we would like to see how the sales at Cape Codd Outdoor Sports are related to the buyers—are the buyers acquiring products that sell? We can start with the SQL-Query-CH02-63:

```
/* *** SQL-Query-CH02-63 *** */
SELECT     OI.OrderNumber, Quantity,
           SD.SKU, SKU_Description, Department, Buyer
FROM       ORDER_ITEM AS OI JOIN SKU_DATA AS SD
    ON     OI.SKU=SD.SKU
ORDER BY   OI.OrderNumber, SD.SKU;
```

This produces the result set:

	OrderNumber	Quantity	SKU	SKU_Description	Department	Buyer
1	1000	1	201000	Half-dome Tent	Camping	Cindy Lo
2	1000	1	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
3	2000	4	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
4	2000	2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
5	3000	1	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
6	3000	2	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
7	3000	1	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers

This result is correct, but it shows the names of only five of the eight SKU items in the SKU_ITEM table. What happened to the other three SKU items and their associated buyers? Look closely at the data in Figure 2-5, and you will see that the three SKU items and their buyers that do not appear in the results (SKU 100100 with buyer Pete Hansen, SKU 301000 with buyer Jerry Martin, and SKU 302000 with buyer Jerry Martin) are SKU items that have never been sold to as part of a retail order. Therefore, the primary key values of these three SKU items

do not match any foreign key value in the ORDER_ITEM, and because they have no match, they do not appear in the result of this join statement. What can we do about this case when we are creating an SQL query?

Consider the STUDENT and LOCKER tables in Figure 2-23(a), where we have drawn two tables to highlight the relationships between the rows in each table. The STUDENT table shows the StudentPK (student number) and StudentName of students at a university. The LOCKER table shows the LockerPK (locker number) and LockerType (full size or half size) of lockers at the recreation center on campus. If we run a standard join between these two tables as shown in SQL-QUERY-CH02-64, we get a table of students who have lockers assigned to them together with their assigned locker. This result is shown in Figure 2-23(b).

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Query-CH02-64 *** */
SELECT     StudentPK, StudentName, LockerFK, LockerPK, LockerType
FROM       STUDENT, LOCKER
WHERE      STUDENT.LockerFK = LOCKER.LockerPK
ORDER BY   StudentPK;
```

The type of SQL join is known as an **SQL inner join**, and we can also run the query using SQL JOIN ON syntax using the **SQL INNER JOIN phrase**. This is shown in SQL QUERY-CH02-65, which produces exactly the same result shown in Figure 2-23(b).

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Query-CH02-65 *** */
SELECT     StudentPK, StudentName, LockerFK, LockerPK, LockerType
FROM       STUDENT INNER JOIN LOCKER
    ON     STUDENT.LockerFK = LOCKER.LockerPK
ORDER BY   StudentPK;
```

Now, suppose we want to show all the rows already in the join, but also want to show any rows (students) in the STUDENT table that are not included in the inner join. This means that we want to see *all students*, including those who have *not been assigned to a locker.* To do this, we use the **SQL outer join**, which is designed for this very purpose. And because the table we want is listed first in the query and is thus on the left side of the table listing, we specifically use an **SQL left outer join**, which uses the **SQL LEFT JOIN syntax**. This is shown in SQL QUERY-CH02-66, which produces the results shown in Figure 2-23(c).

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Query-CH02-66 *** */
SELECT     StudentPK, StudentName, LockerFK, LockerPK, LockerType
FROM       STUDENT LEFT OUTER JOIN LOCKER
    ON     STUDENT.LockerFK = LOCKER.LockerPK
ORDER BY   StudentPK;
```

In the results shown in Figure 2-23(c), note that all the rows from the STUDENT table are now included and that rows that have no match in the LOCKER table are shown with NULL values. Looking at the output, we can see that the students Adams and Buchanan have no linked rows in the LOCKER table. This means that Adams and Buchanan have not been assigned a locker in the recreation center.

If we want to show all the rows already in the join, but now also any rows in the LOCKER table that are not included in the inner join, we specifically use an **SQL right outer join**, which uses the **SQL RIGHT JOIN syntax** because the table we want is listed second in the query and is thus on the right side of the table listing. This means that we want to see *all lockers*,

Figure 2-23
Types of JOINS

STUDENT

StudentPK	StudentName	LockerFK
1	Adams	NULL
2	Buchanan	NULL
3	Carter	10
4	Ford	20
5	Hoover	30
6	Kennedy	40
7	Roosevelt	50
8	Truman	60

LOCKER

LockerPK	LockerType
10	Full
20	Full
30	Half
40	Full
50	Full
60	Half
70	Full
80	Full
90	Half

(a) The STUDENT and LOCKER Tables Aligned to Show Row Relationships

Only the rows where LockerFK=LockerPK are shown—Note that some StudentPK and some LockerPK values are not in the results

StudentPK	StudentName	LockerFK	LockerPK	LockerType
3	Carter	10	10	Full
4	Ford	20	20	Full
5	Hoover	30	30	Half
6	Kennedy	40	40	Full
7	Roosevelt	50	50	Full
8	Truman	60	60	Half

(b) INNER JOIN of the STUDENT and LOCKER Tables

All rows from STUDENT are shown, even where there is no matching LockerFK=LockerPK value

StudentPK	StudentName	LockerFK	LockerPK	LockerType
1	Adams	NULL	NULL	NULL
2	Buchanan	NULL	NULL	NULL
3	Carter	10	10	Full
4	Ford	20	20	Full
5	Hoover	30	30	Half
6	Kennedy	40	40	Full
7	Roosevelt	50	50	Full
8	Truman	60	60	Half

(c) LEFT OUTER JOIN of the STUDENT and LOCKER Tables

All rows from LOCKER are shown, even where there is no matching LockerFK=LockerPK value

StudentPK	StudentName	LockerFK	LockerPK	LockerType
3	Carter	10	10	Full
4	Ford	20	20	Full
5	Hoover	30	30	Half
6	Kennedy	40	40	Full
7	Roosevelt	50	50	Full
8	Truman	60	60	Half
NULL	NULL	NULL	70	Full
NULL	NULL	NULL	80	Full
NULL	NULL	NULL	90	Half

(d) RIGHT OUTER JOIN of the STUDENT and LOCKER Tables

including those that have *not been assigned to a student.* This is shown in SQL QUERY-CH02-67, which produces the results shown in Figure 2-23(d).

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Query-CH02-67 *** */
SELECT     StudentPK, StudentName, LockerFK, LockerPK, LockerType
FROM       STUDENT RIGHT OUTER JOIN LOCKER
    ON     STUDENT.LockerFK
ORDER BY   StudentPK;
```

In the results shown in Figure 2-23(d), note that all the rows from the LOCKER table are now included and that rows that have no match in the STUDENT table are shown with NULL values. Looking at the output, we can see that the lockers numbered 70, 80 and 90 have no linked rows in the STUDENT table. This means that these lockers are currently unassigned to a student, and available for use.

In terms of our question about SKUs and buyers, this means that we can use an SQL OUTER JOIN and specifically an SQL RIGHT OUTER JOIN to obtain the desired results:

```
/* *** SQL-Query-CH02-68 *** */
SELECT     OI.OrderNumber, Quantity,
           SD.SKU, SKU_Description, Department, Buyer
FROM       ORDER_ITEM AS OI RIGHT OUTER JOIN SKU_DATA AS SD
    ON     OI.SKU=SD.SKU
ORDER BY   OI.OrderNumber, SD.SKU;
```

This produces the following results, which clearly show the SKUs and their associated buyers that have not been part of a retail order (in particular, note that we haven't sold any of the 300000 range SKUs, which are climbing equipment—perhaps management should look into that):

	OrderNumber	Quantity	SKU	SKU_Description	Department	Buyer
1	NULL	NULL	100100	Std. Scuba Tank, Yellow	Water Sports	Pete Hansen
2	NULL	NULL	301000	Light Fly Climbing Harness	Climbing	Jerry Martin
3	NULL	NULL	302000	Locking Carabiner, Oval	Climbing	Jerry Martin
4	1000	1	201000	Half-dome Tent	Camping	Cindy Lo
5	1000	1	202000	Half-dome Tent Vestibule	Camping	Cindy Lo
6	2000	4	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
7	2000	2	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers
8	3000	1	100200	Std. Scuba Tank, Magenta	Water Sports	Pete Hansen
9	3000	2	101100	Dive Mask, Small Clear	Water Sports	Nancy Meyers
10	3000	1	101200	Dive Mask, Med Clear	Water Sports	Nancy Meyers

This completes our discussion of SQL query statements. We have covered the needed SQL syntax to allow you to write ad-hoc SQL queries on one or more tables, displaying only the specific row, column, or calculated values that you want to see. In Chapter 7, we will return to SQL to discuss SQL DDL, some other parts of SQL DML, and SQL/PSM.

BY THE WAY

It is easy to forget that inner joins will drop nonmatching rows. Some years ago, one of the authors had a very large organization as a consulting client. The client had a budgetary-planning application that included a long sequence of complicated SQL statements. One of the joins in that sequence was an inner join that should have been an outer join. As a result, some 3,000 employees dropped out of the budgetary calculations. The mistake was discovered only months later when the actual salary expense exceeded the budget salary expense by a large margin. The mistake was an embarrassment all the way to the board of directors.

Summary

Wow! That was a full chapter!

Structured Query Language (SQL) was developed by IBM and has been endorsed by the ANSI SQL-92 and following standards. SQL is a data sublanguage that can be embedded into full programming languages or submitted directly to the DBMS. Knowing SQL is critical for knowledge workers, application programmers, and database administrators.

All DBMS products process SQL. Microsoft Access hides SQL, but Microsoft SQL Server, Oracle Database, and MySQL require that you use it.

We are primarily interested in five categories of SQL statements: DML, DDL, SQL/PSM statements, TCL, and DCL. DML statements include statements for querying data and for inserting, updating, and deleting data. This chapter addresses only DML query statements. Additional DML statements, DDL and SQL/PSM are discussed in Chapter 7. TCL and DCL are discussed in Chapter 9.

The examples in this chapter are based on three tables extracted from the operational database at Cape Codd Outdoor Sports. Such database extracts are common and important. Sample data for the three tables is shown in Figure 2-5.

The basic structure of an SQL query statement is SELECT/FROM/WHERE. The columns to be selected are listed after SELECT, the table(s) to process is listed after FROM, and any restrictions on data values are listed after WHERE. In a WHERE clause, character and date data values must be enclosed in single quotes. Numeric data need not be enclosed in quotes. You can submit SQL statements directly to Microsoft Access, Microsoft SQL Server, Oracle Database, and MySQL, as described in this chapter.

This chapter explained the use of the following SQL clauses: SELECT, FROM, WHERE, ORDER BY, GROUP BY, and HAVING. This chapter explained the use of the following SQL keywords: DISTINCT, DESC, ASC, AND, OR, IN, NOT IN, BETWEEN, LIKE, % (* for Microsoft Access), _ (? for Microsoft Access), SUM, AVG, MIN, MAX, COUNT, and AS. You should know how to mix and match these features to obtain the results you want. By default, the WHERE clause is applied before the HAVING clause.

You can query multiple tables using subqueries and joins. Subqueries are nested queries that use the SQL keywords IN and NOT IN. An SQL SELECT expression is placed inside parentheses. Using a subquery, you can display data from the top table only. A join is created by specifying multiple table names in the FROM clause. An SQL WHERE clause is used to obtain an equijoin. In most cases, equijoins are the most sensible option. Joins can display data from multiple tables. In Chapter 8, you will learn another type of subquery that can perform work that is not possible with joins.

Some people believe the JOIN ON syntax is an easier form of join. Rows that have no match in the join condition are dropped from the join results when using a regular, or INNER, join. To keep such rows, use a LEFT OUTER or RIGHT OUTER join rather than an INNER join.

Key Terms

/* and */
ad-hoc queries
American National Standards Institute (ANSI)
AVG
business intelligence (BI) systems
correlated subquery
COUNT
CRUD
data control language (DCL)
data definition language (DDL)
data manipulation language (DML)
data mart
data sublanguage
data warehouse
data warehouse DBMS
equijoin
Extensible Markup Language (XML)
Extract, Transform, and Load (ETL) system
graphical user interface (GUI)
inner join
International Organization for Standardization (ISO)
join
join operation
joining the two tables
MAX
Microsoft Access asterisk (*) wildcard character
Microsoft Access question mark (?) wildcard character

MIN
query by example (QBE)
schema
SQL AND operator
SQL AS keyword
SQL asterisk (*) wildcard character
SQL BETWEEN keyword
SQL built-in functions
SQL comment
SQL DESC keyword
SQL DISTINCT keyword
SQL expression
SQL FROM clause
SQL GROUP BY clause
SQL HAVING clause
SQL IN operator
SQL inner join
SQL INNER JOIN phrase
SQL JOIN keyword
SQL JOIN ON syntax
SQL join operator
SQL LEFT JOIN syntax
SQL left outer join
SQL LIKE keyword
SQL NOT IN operator
SQL ON keyword
SQL OR operator
SQL ORDER BY clause
SQL outer join
SQL percent sign (%) wildcard character
SQL query
SQL RIGHT JOIN syntax
SQL right outer join
SQL script file
SQL SELECT clause
SQL SELECT/FROM/WHERE framework
SQL Server Compatible Syntax (ANSI 92)
SQL TOP {NumberOfRows} property
SQL underscore (_) wildcard character
SQL WHERE clause
SQL/Persistent stored modules (SQL/PSM)
stock-keeping unit (SKU)
Structured Query Language (SQL)
subquery
SUM
TableName.ColumnName syntax
Transaction control language (TCL)

Review Questions

2.1 What is a business intelligence (BI) system?

2.2 What is an ad-hoc query?

2.3 What does SQL stand for, and what is SQL?

2.4 What does SKU stand for? What is an SKU?

2.5 Summarize how data were altered and filtered in creating the Cape Codd data extraction.

2.6 Explain, in general terms, the relationships among the RETAIL_ORDER, ORDER_ITEM, and SKU_DATA tables.

2.7 Summarize the background of SQL.

2.8 What is SQL-92? How does it relate to the SQL statements in this chapter?

2.9 What features have been added to SQL in versions subsequent to the SQL-92?

2.10 Why is SQL described as a data sublanguage?

2.11 What does DML stand for? What are DML statements?

2.12 What does DDL stand for? What are DDL statements?

2.13 What is the SQL SELECT/FROM/WHERE framework?

2.14 Explain how Microsoft Access uses SQL.

2.15 Explain how enterprise-class DBMS products use SQL.

The Cape Codd Outdoor Sports sale extraction database has been modified to include two additional tables, the INVENTORY table and the WAREHOUSE table. The table schemas for these tables, together with the RETAIL_ORDER, ORDER_ITEM, and SKU_DATA tables, are as follows:

RETAIL_ORDER (OrderNumber, StoreNumber, StoreZip, OrderMonth, OrderYear, OrderTotal)

ORDER_ITEM (*OrderNumber*, *SKU*, Quantity, Price, ExtendedPrice)

SKU_DATA (SKU, SKU_Description, Department, Buyer)

WAREHOUSE (WarehouseID, WarehouseCity, WarehouseState, Manager, Squarefeet)

INVENTORY (*WarehouseID*, *SKU*, SKU_Description, QuantityOnHand, QuantityOnOrder)

The five tables in the revised Cape Codd database schema are shown in Figure 2-24. The column characteristics for the WAREHOUSE table are shown in Figure 2-25, and the column characteristics for the INVENTORY table are shown in Figure 2-26. The data for the WAREHOUSE table are shown in Figure 2-27, and the data for the INVENTORY table are shown in Figure 2-28.

If at all possible, you should run your SQL solutions to the following questions against an actual database. A Microsoft Access database named *Cape-Codd.accdb* is available on our Web site (*www.pearsonhighered.com/kroenke*) that contains all the

Figure 2-24
The Cape Codd Database with the WAREHOUSE and INVENTORY Tables

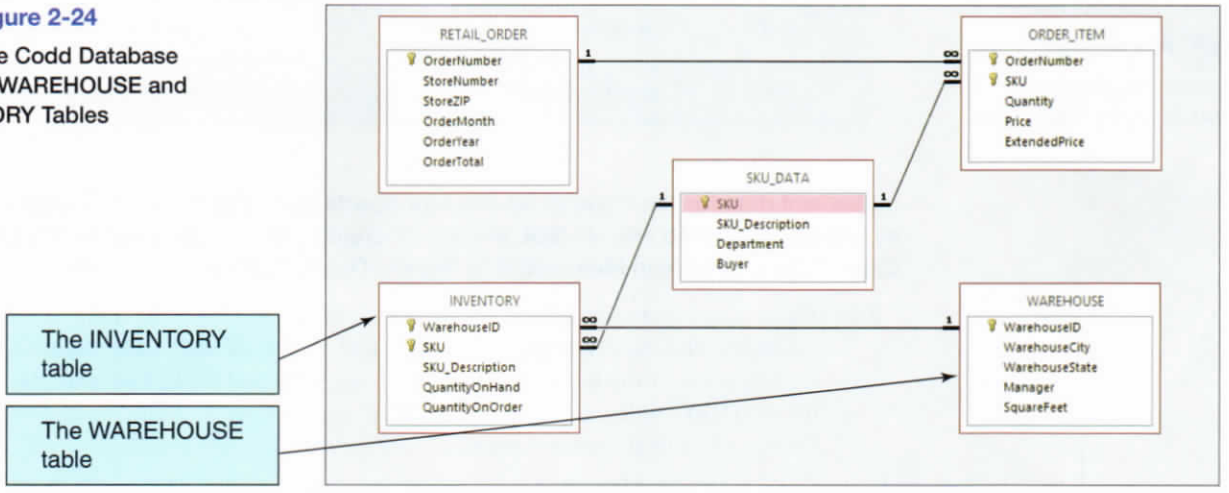


Figure 2-25
Column Characteristics for the Cape Codd Database WAREHOUSE Table

WAREHOUSE

Column Name	Type	Key	Required	Remarks
WarehouseID	Integer	Primary Key	Yes	Surrogate Key
WarehouseCity	Text (30)		Yes	
WarehouseState	Text (2)		Yes	
Manager	Text (35)	No	No	
SquareFeet	Integer	No	No	

INVENTORY

Column Name	Type	Key	Required	Remarks
WarehouseID	Integer	Primary Key, Foreign Key	Yes	Surrogate Key
SKU	Integer	Primary Key, Foreign Key	Yes	Surrogate Key
SKU_Description	Text (35)	No	Yes	
QuantityOnHand	Integer	No	No	
QuantityOnOrder	Integer	No	No	

Figure 2-26
Column Characteristics for the Cape Codd Database INVENTORY Table

WarehouseID	WarehouseCity	WarehouseState	Manager	SquareFeet
100	Atlanta	GA	Dave Jones	125,000
200	Chicago	IL	Lucille Smith	100,000
300	Bangor	MA	Bart Evans	150,000
400	Seattle	WA	Dale Rogers	130,000
500	San Francisco	CA	Grace Jefferson	200,000

Figure 2-27
Cape Codd Database WAREHOUSE Table Data

tables and data for the Cape Codd Outdoor Sports sales data extract database. Also available on our Web site are SQL scripts for creating and populating the tables for the Cape Codd database in Microsoft SQL Server, Oracle Database, and MySQL.

2.16 There is an intentional flaw in the design of the INVENTORY table used in these exercises. This flaw was purposely included in the INVENTORY tables so you can answer some of the following questions using only that table. Compare the SKU and INVENTORY tables, and determine what design flaw is included in INVENTORY. Specifically, why did we include it?

Use *only* the INVENTORY table to answer Review Questions 2.17 through 2.39:

2.17 Write an SQL statement to display SKU and SKU_Description.

2.18 Write an SQL statement to display SKU_Description and SKU.

2.19 Write an SQL statement to display WarehouseID.

2.20 Write an SQL statement to display unique WarehouseIDs.

2.21 Write an SQL statement to display all of the columns without using the SQL asterisk (*) wildcard character.

2.22 Write an SQL statement to display all of the columns using the SQL asterisk (*) wildcard character.

2.23 Write an SQL statement to display all data on products having a QuantityOnHand greater than 0.

2.24 Write an SQL statement to display the SKU and SKU_Description for products having QuantityOnHand equal to 0.

WarehouseID	SKU	SKU_Description	QuantityOnHand	QuantityOnOrder
100	100100	Std. Scuba Tank, Yellow	250	0
200	100100	Std. Scuba Tank, Yellow	100	50
300	100100	Std. Scuba Tank, Yellow	100	0
400	100100	Std. Scuba Tank, Yellow	200	0
100	100200	Std. Scuba Tank, Magenta	200	30
200	100200	Std. Scuba Tank, Magenta	75	75
300	100200	Std. Scuba Tank, Magenta	100	100
400	100200	Std. Scuba Tank, Magenta	250	0
100	101100	Dive Mask, Small Clear	0	500
200	101100	Dive Mask, Small Clear	0	500
300	101100	Dive Mask, Small Clear	300	200
400	101100	Dive Mask, Small Clear	450	0
100	101200	Dive Mask, Med Clear	100	500
200	101200	Dive Mask, Med Clear	50	500
300	101200	Dive Mask, Med Clear	475	0
400	101200	Dive Mask, Med Clear	250	250
100	201000	Half-Dome Tent	2	100
200	201000	Half-Dome Tent	10	250
300	201000	Half-Dome Tent	250	0
400	201000	Half-Dome Tent	0	250
100	202000	Half-Dome Tent Vestibule	10	250
200	202000	Half-Dome Tent Vestibule	1	250
300	202000	Half-Dome Tent Vestibule	100	0
400	202000	Half-Dome Tent Vestibule	0	200
100	301000	Light Fly Climbing Harness	300	250
200	301000	Light Fly Climbing Harness	250	250
300	301000	Light Fly Climbing Harness	0	250
400	301000	Light Fly Climbing Harness	0	250
100	302000	Locking Carabiner, Oval	1000	0
200	302000	Locking Carabiner, Oval	1250	0
300	302000	Locking Carabiner, Oval	500	500
400	302000	Locking Carabiner, Oval	0	1000

Figure 2-28
Cape Codd Database
INVENTORY Table Data

2.25 Write an SQL statement to display the SKU, SKU_Description, and WarehouseID for products that have a QuantityOnHand equal to 0. Sort the results in ascending order by WarehouseID.

2.26 Write an SQL statement to display the SKU, SKU_Description, and WarehouseID for products that have a QuantityOnHand greater than 0. Sort the results in descending order by WarehouseID and in ascending order by SKU.

2.27 Write an SQL statement to display SKU, SKU_Description, and WarehouseID for all products that have a QuantityOnHand equal to 0 and a QuantityOnOrder greater than 0. Sort the results in descending order by WarehouseID and in ascending order by SKU.

2.28 Write an SQL statement to display SKU, SKU_Description, and WarehouseID for all products that have a QuantityOnHand equal to 0 or a QuantityOnOrder equal to 0. Sort the results in descending order by WarehouseID and in ascending order by SKU.

2.29 Write an SQL statement to display the SKU, SKU_Description, WarehouseID, and QuantityOnHand for all products having a QuantityOnHand greater than 1 and less than 10. Do not use the BETWEEN keyword.

2.30 Write an SQL statement to display the SKU, SKU_Description, WarehouseID, and QuantityOnHand for all products having a QuantityOnHand greater than 1 and less than 10. Use the BETWEEN keyword.

2.31 Write an SQL statement to show a unique SKU and SKU_Description for all products having an SKU description starting with 'Half-dome'.

2.32 Write an SQL statement to show a unique SKU and SKU_Description for all products having a description that includes the word 'Climb'.

2.33 Write an SQL statement to show a unique SKU and SKU_Description for all products having a 'd' in the third position from the left in SKU_Description.

2.34 Write an SQL statement that uses all of the SQL built-in functions on the QuantityOnHand column. Include meaningful column names in the result.

2.35 Explain the difference between the SQL built-in functions COUNT and SUM.

2.36 Write an SQL statement to display the WarehouseID and the sum of QuantityOnHand, grouped by WarehouseID. Name the sum TotalItemsOnHand. Display the results in descending order of TotalItemsOnHand.

2.37 Write an SQL statement to display the WarehouseID and the sum of QuantityOnHand, grouped by WarehouseID. Omit all SKU items that have 3 or more items on hand from the sum, and name the sum TotalItemsOnHandLT3. Display the results in descending order of TotalItemsOnHandLT3.

2.38 Write an SQL statement to display the WarehouseID and the sum of QuantityOnHand, grouped by WarehouseID. Omit all SKU items that have 3 or more items on hand from the sum, and name the sum TotalItemsOnHandLT3. Show Warehouse ID only for warehouses having fewer than 2 SKUs in their TotalItemsOnHandLT3. Display the results in descending order of TotalItemsOnHandLT3.

2.39 In your answer to Review Question 2.38, was the WHERE clause or the HAVING clause applied first? Why?

Use *both* the INVENTORY and WAREHOUSE tables to answer Review Questions 2.40 through 2.55:

2.40 Write an SQL statement to display the SKU, SKU_Description, WarehouseID, WarehouseCity, and WarehouseState for all items stored in the Atlanta, Bangor, or Chicago warehouse. Do not use the IN keyword.

2.41 Write an SQL statement to display the SKU, SKU_Description, WarehouseID, WarehouseCity, and WarehouseState for all items stored in the Atlanta, Bangor, or Chicago warehouse. Use the IN keyword.

2.42 Write an SQL statement to display the SKU, SKU_Description, WarehouseID, WarehouseCity, and WarehouseState of all items not stored in the Atlanta, Bangor, or Chicago warehouse. Do not use the NOT IN keyword.

2.43 Write an SQL statement to display the SKU, SKU_Description, WarehouseID, WarehouseCity, and WarehouseState of all items not stored in the Atlanta, Bangor, or Chicago warehouse. Use the NOT IN keyword.

2.44 Write an SQL statement to produce a single column called ItemLocation that combines the SKU_Description, the phrase "is in a warehouse in", and WarehouseCity. Do not be concerned with removing leading or trailing blanks.

2.45 Write an SQL statement to show the SKU, SKU_Description, and WarehouseID for all items stored in a warehouse managed by 'Lucille Smith'. Use a subquery.

2.46 Write an SQL statement to show the SKU, SKU_Description, and WarehouseID for all items stored in a warehouse managed by 'Lucille Smith'. Use a join, but do not use JOIN ON syntax.

2.47 Write an SQL statement to show the SKU, SKU_Description, and WarehouseID for all items stored in a warehouse managed by 'Lucille Smith'. Use a join using JOIN ON syntax.

2.48 Write an SQL statement to show the WarehouseID and average QuantityOnHand of all items stored in a warehouse managed by 'Lucille Smith'. Use a subquery.

2.49 Write an SQL statement to show the WarehouseID and average QuantityOnHand of all items stored in a warehouse managed by 'Lucille Smith'. Use a join, but do not use JOIN ON syntax.

2.50 Write an SQL statement to show the WarehouseID and average QuantityOnHand of all items stored in a warehouse managed by 'Lucille Smith'. Use a join using JOIN ON syntax.

2.51 Write an SQL statement to display the WarehouseID, the sum of QuantityOnOrder, and the sum of QuantityOnHand, grouped by WarehouseID and QuantityOnOrder. Name the sum of QuantityOnOrder as TotalItemsOnOrder and the sum of QuantityOnHand as TotalItemsOnHand.

2.52 Write an SQL statement to show the WarehouseID, WarehouseCity, WarehouseState, Manager, SKU, SKU_Description, and QuantityOnHand of all items with a Manager of 'Lucille Smith'. Use a join.

2.53 Explain why you cannot use a subquery in your answer to Review Question 2.51.

2.54 Explain how subqueries and joins differ.

2.55 Write an SQL statement to join WAREHOUSE and INVENTORY and include all rows of WAREHOUSE in your answer, regardless of whether they have any INVENTORY. Run this statement.

Project Questions

For this set of project questions, we will extend the Microsoft Access database for the Wedgewood Pacific Corporation (WPC) that we created in Chapter 1. Founded in 1957 in Seattle, Washington, WPC has grown into an internationally recognized organization. The company is located in two buildings. One building houses the Administration, Accounting, Finance, and Human Resources departments, and the second houses the Production, Marketing, and Information Systems departments.

The company database contains data about company employees, departments, company projects, company assets such as computer equipment, and other aspects of company operations.

In the following project questions, we have already created the WPC.accdb database with the following two tables (see Chapter 1 Project Questions):

DEPARTMENT (DepartmentName, BudgetCode, OfficeNumber, Phone)

EMPLOYEE (EmployeeNumber, FirstName, LastName, *Department*, Phone, Email)

Now we will add in the following two tables:

PROJECT (ProjectID, Name, *Department*, MaxHours, StartDate, EndDate)

ASSIGNMENT (*ProjectID*, *EmployeeNumber*, HoursWorked)

The four tables in the revised WPC database schema are shown in Figure 2-29. The column characteristics for the PROJECT table are shown in Figure 2-30, and the column characteristics for the ASSIGNMENT table are shown in Figure 2-32. Data for the PROJECT table are shown in Figure 2-31, and the data for the ASSIGNMENT table are shown in Figure 2-33.

2.56 Figure 2-30 shows the column characteristics for the WPC PROJECT table. Using the column characteristics, create the PROJECT table in the WPC.accdb database.

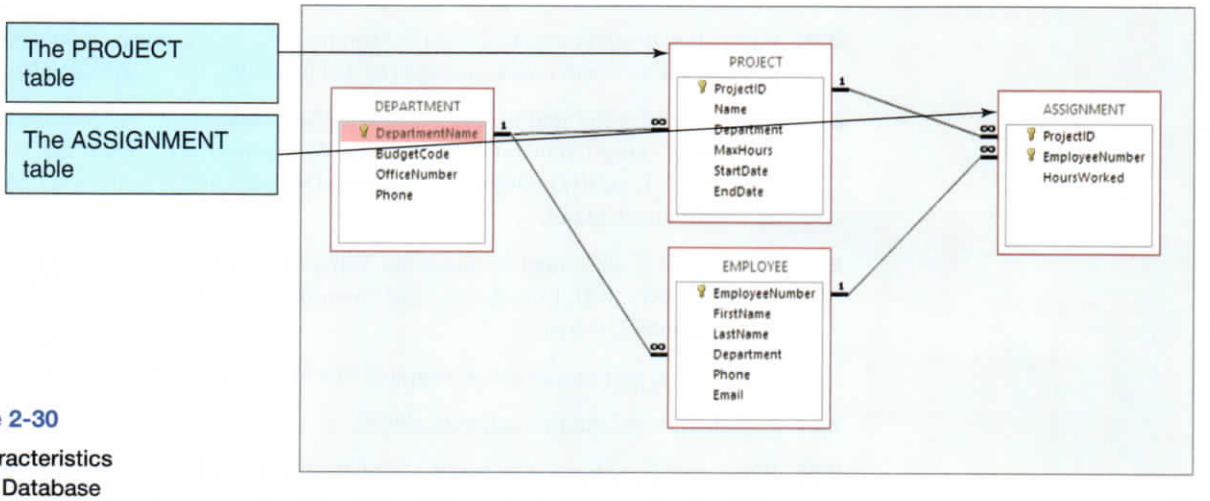


Figure 2-29
The WPC Database with the PROJECT and ASSIGNMENT Tables

Figure 2-30
Column Characteristics for the WPC Database PROJECT Table

PROJECT

Column Name	Type	Key	Required	Remarks
ProjectID	Number	Primary Key	Yes	Long Integer
Name	Text (50)	No	Yes	
Department	Text (35)	Foreign Key	Yes	
MaxHours	Number	No	Yes	Double
StartDate	Date	No	No	
EndDate	Date	No	No	

ProjectID	Name	Department	MaxHours	StartDate	EndDate
1000	2013 Q3 Product Plan	Marketing	135.00	10-MAY-13	15-JUN-13
1100	2013 Q3 Portfolio Analysis	Finance	120.00	07-JUL-13	25-JUL-13
1200	2013 Q3 Tax Preparation	Accounting	145.00	10-AUG-13	15-OCT-13
1300	2013 Q4 Product Plan	Marketing	150.00	10-AUG-13	15-SEP-13
1400	2013 Q4 Portfolio Analysis	Finance	140.00	05-OCT-13	

Figure 2-31
Sample Data for the WPC Database PROJECT Table

ASSIGNMENT

Column Name	Type	Key	Required	Remarks
ProjectID	Number	Primary Key, Foreign Key	Yes	Long Integer
EmployeeNumber	Number	Primary Key, Foreign Key	Yes	Long Integer
HoursWorked	Number	No	No	Double

Figure 2-32
Column Characteristics for the WPC Database ASSIGNMENT Table

ProjectID	EmployeeNumber	HoursWorked
1000	1	30.0
1000	8	75.0
1000	10	55.0
1100	4	40.0
1100	6	45.0
1100	1	25.0
1200	2	20.0
1200	4	45.0
1200	5	40.0
1300	1	35.0
1300	8	80.0
1300	10	50.0
1400	4	15.0
1400	5	10.0
1400	6	27.5

Figure 2-33
Sample Data for the WPC Database ASSIGNMENT Table

2.57 Create the relationship and referential integrity constraint between PROJECT and DEPARTMENT. Enable enforcing of referential integrity and cascading of data updates, but do *not* enable cascading of data from deleted records.

2.58 Figure 2-31 shows the data for the WPC PROJECT table. Using the Datasheet view, enter the data shown in Figure 2-31 into your PROJECT table.

2.59 Figure 2-32 shows the column characteristics for the WPC ASSIGNMENT table. Using the column characteristics, create the ASSIGNMENT table in the WPC.accdb database.

2.60 Create the relationship and referential integrity constraint between ASSIGNMENT and EMPLOYEE. Enable enforcing of referential integrity, but do *not* enable either cascading updates or the cascading of data from deleted records.

2.61 Create the relationship and referential integrity constraint between ASSIGNMENT and PROJECT. Enable enforcing of referential integrity and cascading of deletes, but do *not* enable cascading updates.

2.62 Figure 2-33 shows the data for the WPC ASSIGNMENT table. Using the Datasheet view, enter the data shown in Figure 2-33 into your ASSIGNMENT table.

2.63 In Review Question 2.58, the table data was entered after referential integrity constraints were created in Review Question 2.57. In Review Question 2.62, the table data was entered after referential integrity constraints were created in Review Questions 2.59 and 2.60. Why was the data entered after the referential integrity constraints were created instead of before the constraints were created?

2.64 Using Microsoft Access SQL, create and run queries to answer the following questions. Save each query using the query name format SQL-Query-02-##, where the ## sign is replaced by the letter designator of the question. For example, the first query will be saved as SQL-Query-02-A.

A. What projects are in the PROJECT table? Show all information for each project.

B. What are the ProjectID, Name, StartDate, and EndDate values of projects in the PROJECT table?

C. What projects in the PROJECT table started before August 1, 2013? Show all the information for each project.

D. What projects in the PROJECT table have not been completed? Show all the information for each project.

E. Who are the employees assigned to each project? Show ProjectID, EmployeeNumber, LastName, FirstName, and Phone.

F. Who are the employees assigned to each project? The ProjectID, Name, and Department. Show EmployeeNumber, LastName, FirstName, and Phone.

G. Who are the employees assigned to each project? Show ProjectID, Name, Department, and Department Phone. Show EmployeeNumber, LastName, FirstName, and Employee Phone. Sort by ProjectID, in ascending order.

H. Who are the employees assigned to projects run by the marketing department? Show ProjectID, Name, Department, and Department Phone. Show EmployeeNumber, LastName, FirstName, and Employee Phone. Sort by ProjectID, in ascending order.

I. How many projects are being run by the marketing department? Be sure to assign an appropriate column name to the computed results.

J. What is the total MaxHours of projects being run by the marketing department? Be sure to assign an appropriate column name to the computed results.

K. What is the average MaxHours of projects being run by the marketing department? Be sure to assign an appropriate column name to the computed results.

L. How many projects are being run by each department? Be sure to display each DepartmentName and to assign an appropriate column name to the computed results.

M. Write an SQL statement to join EMPLOYEE, ASSIGNMENT, and PROJECT using the JOIN ON syntax. Run this statement.

N. Write an SQL statement to join EMPLOYEE and ASSIGNMENT and include all rows of EMPLOYEE in your answer, regardless of whether they have an ASSIGNMENT. Run this statement.

2.65 Using Microsoft Access QBE, create and run new queries to answer the questions in Project Question 2.64. Save each query using the query name format QBE-Query-02-##, where the ## sign is replaced by the letter designator of the question. For example, the first query will be saved as QBE-Query-02-A.

The following questions refer to the NDX table data as described starting on page 74. You can obtain a copy of this data in the Microsoft Access database *DBP-e13-NDX .accdb* from the text's Web site (*www.pearsonhighered.com/kroenke*).

2.66 Write SQL queries to produce the following results:

A. The ChangeClose on Fridays.

B. The minimum, maximum, and average ChangeClose on Fridays.

C. The average ChangeClose grouped by TYear. Show TYear.

D. The average ChangeClose grouped by TYear and TMonth. Show TYear and TMonth.

E. The average ChangeClose grouped by TYear, TQuarter, TMonth shown in descending order of the average (you will have to give a name to the average in order to sort by it). Show TYear, TQuarter, and TMonth. Note that months appear in alphabetical and not calendar order. Explain what you need to do to obtain months in calendar order.

F. The difference between the maximum ChangeClose and the minimum ChangeClose grouped by TYear, TQuarter, TMonth shown in descending order of the difference (you will have to give a name to the difference in order to sort by it). Show TYear, TQuarter, and TMonth.

G. The average ChangeClose grouped by TYear shown in descending order of the average (you will have to give a name to the average in order to sort by it). Show only groups for which the average is positive.

H. Display a single field with the date in the form day/month/year. Do not be concerned with trailing blanks.

2.67 It is possible that volume (the number of shares traded) has some correlation with the direction of the stock market. Use the SQL you have learned in this chapter to investigate this possibility. Develop at least five different SQL statements in your investigation.

Case Questions

Marcia's Dry Cleaning Case Questions

Marcia Wilson owns and operates *Marcia's Dry Cleaning*, which is an upscale dry cleaner in a well-to-do suburban neighborhood. Marcia makes her business stand out from the competition by providing superior customer service. She wants to keep track of each of her customers and their orders. Ultimately, she wants to notify them that their clothes are ready via e-mail. To

provide this service, she has developed an initial database with several tables. Three of those tables are the following:

CUSTOMER (CustomerID, FirstName, LastName, Phone, Email)
INVOICE (InvoiceNumber, *CustomerNumber*, DateIn, DateOut, TotalAmount)
INVOICE_ITEM (*InvoiceNumber*, ItemNumber, Item, Quantity, UnitPrice)

In the database schema above, the primary keys are underlined and the foreign keys are shown in italics. The database that Marcia has created is named MDC, and the three tables in the MDC database schema are shown in Figure 2-34.

The column characteristics for the tables are shown in Figures 2-35, 2-36, and 2-37. The relationship between CUSTOMER and INVOICE should enforce referential integrity, but not cascade updates nor deletions, while the relationship between INVOICE and INVOICE_ITEM should enforce referential integrity and cascade both updates and deletions. The data for these tables are shown in Figures 2-38, 2-39, and 2-40.

We recommend that you create a Microsoft Access 2013 database named *MDC-CH02.accdb* using the database schema, column characteristics, and data shown above and then use this database to test your solutions to the questions in this section. Alternatively, SQL scripts for creating the *MDC-CH02* database in Microsoft SQL Server, Oracle Database, and MySQL are available on our Web site at *www.pearsonhighered.com/kroenke*.

Write SQL statements and show the results based on the MDC data for each of the following:

A. Show all data in each of the tables.

B. List the LastName, FirstName, and Phone of all customers.

Figure 2-34
The MDC Database

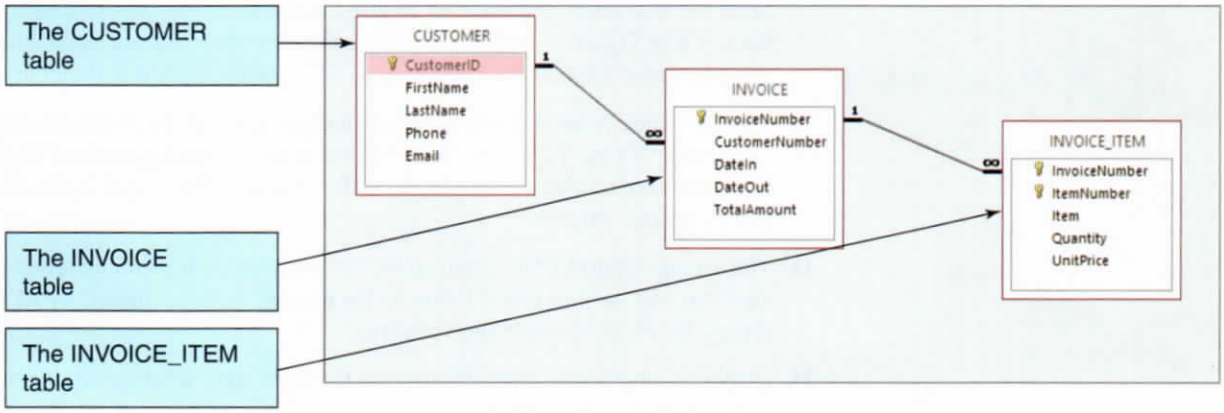


Figure 2-35
Column Characteristics for the MDC Database CUSTOMER Table

CUSTOMER

Column Name	Type	Key	Required	Remarks
CustomerID	AutoNumber	Primary Key	Yes	Surrogate Key
FirstName	Text (25)	No	Yes	
LastName	Text (25)	No	Yes	
Phone	Text (12)	No	No	
Email	Text (100)	No	No	

INVOICE

Column Name	Type	Key	Required	Remarks
InvoiceNumber	Number	Primary Key	Yes	Long Integer
CustomerNumber	Number	Foreign Key	Yes	Long Integer
DateIn	Date	No	Yes	
DateOut	Date	No	No	
TotalAmount	Currency	No	No	Two Decimal Places

Figure 2-36
Column Characteristics for the MDC Database INVOICE Table

INVOICE_ITEM

Column Name	Type	Key	Required	Remarks
InvoiceNumber	Number	Primary Key, Foreign Key	Yes	Long Integer
ItemNumber	Number	Primary Key	Yes	Long Integer
Item	Text (50)	No	Yes	
Quantity	Number	No	Yes	Long Integer
UnitPrice	Currency	No	Yes	Two Decimal Places

Figure 2-37
Column Characteristics for the MDC Database INVOICE_ITEM Table

C. List the LastName, FirstName, and Phone for all customers with a FirstName of 'Nikki'.

D. List the LastName, FirstName, Phone, DateIn, and DateOut of all orders in excess of $100.00.

E. List the LastName, FirstName, and Phone of all customers whose first name starts with 'B'.

F. List the LastName, FirstName, and Phone of all customers whose last name includes the characters 'cat'.

Figure 2-38
Sample Data for the MDC Database CUSTOMER Table

CustomerID	FirstName	LastName	Phone	Email
1	Nikki	Kaccaton	723-543-1233	Nikki.Kaccaton@somewhere.com
2	Brenda	Catnazaro	723-543-2344	Brenda.Catnazaro@somewhere.com
3	Bruce	LeCat	723-543-3455	Bruce.LeCat@somewhere.com
4	Betsy	Miller	725-654-3211	Betsy.Miller@somewhere.com
5	George	Miller	725-654-4322	George.Miller@somewhere.com
6	Kathy	Miller	723-514-9877	Kathy.Miller@somewhere.com
7	Betsy	Miller	723-514-8766	Betsy.Miller@elsewhere.com

InvoiceNumber	CustomerNumber	DateIn	DateOut	TotalAmount
2013001	1	04-Oct-13	06-Oct-13	$158.50
2013002	2	04-Oct-13	06-Oct-13	$25.00
2013003	1	06-Oct-13	08-Oct-13	$49.00
2013004	4	06-Oct-13	08-Oct-13	$17.50
2013005	6	07-Oct-13	11-Oct-13	$12.00
2013006	3	11-Oct-13	13-Oct-13	$152.50
2013007	3	11-Oct-13	13-Oct-13	$7.00
2013008	7	12-Oct-13	14-Oct-13	$140.50
2013009	5	12-Oct-13	14-Oct-13	$27.00

Figure 2-39
Sample Data for the MDC Database INVOICE Table

G. List the LastName, FirstName, and Phone for all customers whose second and third numbers (from the right) of their phone number are 23.

H. Determine the maximum and minimum TotalAmount.

I. Determine the average TotalAmount.

J. Count the number of customers.

K. Group customers by LastName and then by FirstName.

L. Count the number of customers having each combination of LastName and FirstName.

M. Show the LastName, FirstName, and Phone of all customers who have had an order with TotalAmount greater than $100.00. Use a subquery. Present the results sorted by LastName in ascending order and then FirstName in descending order.

N. Show the LastName, FirstName, and Phone of all customers who have had an order with TotalAmount greater than $100.00. Use a join, but do not use JOIN ON syntax. Present results sorted by LastName in ascending order and then FirstName in descending order.

O. Show the LastName, FirstName, and Phone of all customers who have had an order with TotalAmount greater than $100.00. Use a join using JOIN ON syntax. Present results sorted by LastName in ascending order and then FirstName in descending order.

P. Show the LastName, FirstName, and Phone of all customers who have had an order with an Item named 'Dress Shirt'. Use a subquery. Present results sorted by LastName in ascending order and then FirstName in descending order.

Q. Show the LastName, FirstName, and Phone of all customers who have had an order with an Item named 'Dress Shirt'. Use a join, but do not use JOIN ON syntax. Present results sorted by LastName in ascending order and then FirstName in descending order.

R. Show the LastName, FirstName, and Phone of all customers who have had an order with an Item named 'Dress Shirt'. Use a join using JOIN ON syntax. Present results sorted by LastName in ascending order and then FirstName in descending order.

InvoiceNumber	ItemNumber	Item	Quantity	UnitPrice
2013001	1	Blouse	2	$3.50
2013001	2	Dress Shirt	5	$2.50
2013001	3	Formal Gown	2	$10.00
2013001	4	Slacks-Mens	10	$5.00
2013001	5	Slacks-Womens	10	$6.00
2013001	6	Suit-Mens	1	$9.00
2013002	1	Dress Shirt	10	$2.50
2013003	1	Slacks-Mens	5	$5.00
2013003	2	Slacks-Womens	4	$6.00
2013004	1	Dress Shirt	7	$2.50
2013005	1	Blouse	2	$3.50
2013005	2	Dress Shirt	2	$2.50
2013006	1	Blouse	5	$3.50
2013006	2	Dress Shirt	10	$2.50
2013006	3	Slacks-Mens	10	$5.00
2013006	4	Slacks-Womens	10	$6.00
2013007	1	Blouse	2	$3.50
2013008	1	Blouse	3	$3.50
2013008	2	Dress Shirt	12	$2.50
2013008	3	Slacks-Mens	8	$5.00
2013008	4	Slacks-Womens	10	$6.00
2013009	1	Suit-Mens	3	$9.00

Figure 2-40
Sample Data for the MDC Database INVOICE_ITEM Table

S. Show the LastName, FirstName, Phone, and TotalAmount of all customers who have had an order with an Item named 'Dress Shirt'. Use a combination of a join and a subquery. Present results sorted by LastName in ascending order and then FirstName in descending order.

T. Show the LastName, FirstName, Phone, and TotalAmount of all customers who have had an order with an Item named 'Dress Shirt'. Also show the LastName, FirstName, and Phone of all other customers. Present results sorted by LastName in ascending order, and then FirstName in descending order.

The Queen Anne Curiosity Shop

The Queen Anne Curiosity Shop is an upscale home furnishings store in a well-to-do urban neighborhood. It sells both antiques and current-production household items that complement or are useful with the antiques. For example, the store sells antique dining room tables and new tablecloths. The antiques are purchased from both individuals and wholesalers, and the new items are purchased from distributors. The store's customers include individuals, owners of bed-and-breakfast operations, and local interior designers who work with both individuals and small businesses. The antiques are unique, though some multiple items, such as dining room chairs, may be available as a set (sets are never broken). The new items are not unique, and an item may be reordered if it is out of stock. New items are also available in various sizes and colors (for example, a particular style of tablecloth may be available in several sizes and in a variety of colors).

Assume that The Queen Anne Curiosity Shop designs a database with the following tables:

CUSTOMER (CustomerID, LastName, FirstName, Address, City, State, ZIP, Phone, Email)

ITEM (ItemID, ItemDescription, CompanyName, PurchaseDate, ItemCost, ItemPrice)

SALE (SaleID, *CustomerID*, SaleDate, SubTotal, Tax, Total)

SALE_ITEM (*SaleID*, SaleItemID, *ItemID*, ItemPrice)

The referential integrity constraints are:

CustomerID in SALE must exist in CustomerID in CUSTOMER

SaleID in SALE_ITEM must exist in SaleID in SALE

ItemID in SALE_ITEM must exist in ItemID in ITEM

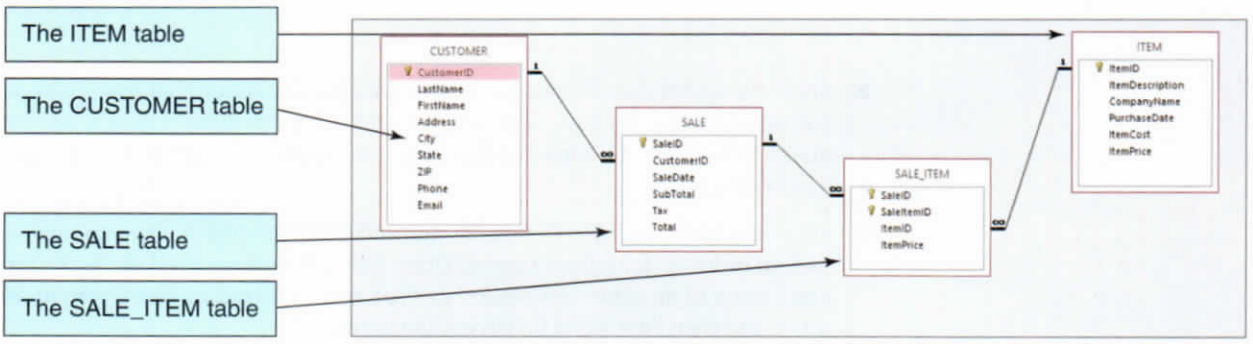
Assume that CustomerID of CUSTOMER, ItemID of ITEM, SaleID of SALE, and SaleItemID of SALE_ITEM are all surrogate keys with values as follows:

CustomerID	Start at 1	Increment by 1
ItemID	Start at 1	Increment by 1
SaleID	Start at 1	Increment by 1

The database that The Queen Anne Curiosity Shop has created is named QACS, and the four tables in the QACS database schema are shown in Figure 2-41.

The column characteristics for the tables are shown in Figures 2-42, 2-43, 2-44, and 2-45. The relationships CUSTOMER-to-SALE and ITEM-to-SALE_ITEM should enforce referential integrity, but not cascade updates nor deletions, while the relationship between SALE and SALE_ITEM should enforce referential integrity and cascade both updates and deletions. The data for these tables are shown in Figures 2-46, 2-47, 2-48, and 2-49.

Figure 2-41
The QACS Database



CUSTOMER

Column Name	Type	Key	Required	Remarks
CustomerID	AutoNumber	Primary Key	Yes	Surrogate Key
LastName	Text (25)	No	Yes	
FirstName	Text (25)	No	Yes	
Address	Text (35)	No	No	
City	Text (35)	No	No	
State	Text (2)	No	No	
ZIP	Text (10)	No	No	
Phone	Text (12)	No	Yes	
Email	Text (100)	No	Yes	

Figure 2-42
Column Characteristics for the QACS Database CUSTOMER Table

Figure 2-43
Column Characteristics for the QACS Database SALE Table

SALE

Column Name	Type	Key	Required	Remarks
SaleID	AutoNumber	Primary Key	Yes	Surrogate Key
CustomerID	Number	Foreign Key	Yes	Long Integer
SaleDate	Date	No	Yes	
SubTotal	Number	No	No	Currency, 2 decimal places
Tax	Number	No	No	Currency, 2 decimal places
Total	Number	No	No	Currency, 2 decimal places

Figure 2-44
Column Characteristics for the QACS Database SALE_ITEM Table

SALE_ITEM

Column Name	Type	Key	Required	Remarks
SaleID	Number	Primary Key, Foreign Key	Yes	Long Integer
SaleItemID	Number	Primary Key	Yes	Long Integer
ItemID	Number	Number	Yes	Long Integer
ItemPrice	Number	No	No	Currency, 2 decimal places

ITEM

Column Name	Type	Key	Required	Remarks
ItemID	AutoNumber	Primary Key	Yes	Surrogate Key
ItemDescription	Text (255)	No	Yes	
CompanyName	Text (100)	No	Yes	
PurchaseDate	Date	No	Yes	
ItemCost	Number	No	Yes	Currency, 2 decimal places
ItemPrice	Number	No	Yes	Currency, 2 decimal places

Figure 2-45
Column Characteristics for the QACS Database ITEM Table

We recommend that you create a Microsoft Access 2013 database named *QACS-CH02.accdb* using the database schema, column characteristics, and data shown above and then use this database to test your solutions to the questions in this section. Alternatively, SQL scripts for creating the *QACS-CH02* database in Microsoft SQL Server, Oracle Database, and MySQL are available on our Web site at *www.pearsonhighered.com/kroenke*.

Write SQL statements and show the results based on the QACS data for each of the following:

A. Show all data in each of the tables.

B. List the LastName, FirstName, and Phone of all customers.

C. List the LastName, FirstName, and Phone for all customers with a FirstName of 'John'.

D. List the LastName, FirstName, and Phone of all customers with a last name of 'Anderson'.

E. List the LastName, FirstName, and Phone of all customers whose first name starts with 'D'.

F. List the LastName, FirstName, and Phone of all customers whose last name includes the characters 'ne'.

G. List the LastName, FirstName, and Phone for all customers whose second and third numbers (from the right) of their phone number are 56.

H. Determine the maximum and minimum sales Total.

I. Determine the average sales Total.

J. Count the number of customers.

K. Group customers by LastName and then by FirstName.

L. Count the number of customers having each combination of LastName and FirstName.

M. Show the LastName, FirstName, and Phone of all customers who have had an order with Total greater than $100.00. Use a subquery. Present the results sorted by LastName in ascending order and then FirstName in descending order.

CustomerID	LastName	FirstName	Address	City	State	ZIP	Phone	Email
1	Shire	Robert	6225 Evanston Ave N	Seattle	WA	98103	206-524-2433	Rober.Shire@somewhere.com
2	Goodyear	Katherine	7335 11th Ave NE	Seattle	WA	98105	206-524-3544	Katherine.Goodyear@somewhere.com
3	Bancroft	Chris	12605 NE 6th Street	Bellevue	WA	98005	425-635-9788	Chris.Bancroft@somewhere.com
4	Griffith	John	335 Aloha Street	Seattle	WA	98109	206-524-4655	John.Griffith@somewhere.com
5	Tierney	Doris	14510 NE 4th Street	Bellevue	WA	98005	425-635-8677	Doris.Tierney@somewhere.com
6	Anderson	Donna	1410 Hillcrest Parkway	Mt. Vernon	WA	98273	360-538-7566	Donna.Anderson@elsewhere.com
7	Svane	Jack	3211 42nd Street	Seattle	WA	98115	206-524-5766	Jack.Svane@somewhere.com
8	Walsh	Denesha	6712 24th Avenue NE	Redmond	WA	98053	425-635-7566	Denesha.Walsh@somewhere.com
9	Enquist	Craig	534 15th Street	Bellingham	WA	98225	360-538-6455	Craig.Enquist@elsewhere.com
10	Anderson	Rose	6823 17th Ave NE	Seattle	WA	98105	206-524-6877	Rose.Anderson@elsewhere.com

Figure 2-46
Sample Data for the QACS Database CUSTOMER Table

SaleID	CustomerID	SaleDate	SubTotal	Tax	Total
1	1	12/14/2012	$3,500.00	$290.50	$3,790.50
2	2	12/15/2012	$1,000.00	$83.00	$1,083.00
3	3	12/15/2012	$50.00	$4.15	$54.15
4	4	12/23/2012	$45.00	$3.74	$48.74
5	1	1/5/2013	$250.00	$20.75	$270.75
6	5	1/10/2013	$750.00	$62.25	$812.25
7	6	1/12/2013	$250.00	$20.75	$270.75
8	2	1/15/2013	$3,000.00	$249.00	$3,249.00
9	5	1/25/2013	$350.00	$29.05	$379.05
10	7	2/4/2013	$14,250.00	$1,182.75	$15,432.75
11	8	2/4/2013	$250.00	$20.75	$270.75
12	5	2/7/2013	$50.00	$4.15	$54.15
13	9	2/7/2013	$4,500.00	$373.50	$4,873.50
14	10	2/11/2013	$3,675.00	$305.03	$3,980.03
15	2	2/11/2013	$800.00	$66.40	$866.40

Figure 2-47
Sample Data for the QACS Database SALE Table

N. Show the LastName, FirstName, and Phone of all customers who have had an order with Total greater than $100.00. Use a join, but do not use JOIN ON syntax. Present results sorted by LastName in ascending order and then FirstName in descending order.

O. Show the LastName, FirstName, and Phone of all customers who have had an order with Total greater than $100.00. Use a join using JOIN ON syntax. Present results sorted by LastName in ascending order and then FirstName in descending order.

P. Show the LastName, FirstName, and Phone of all customers who who have bought an Item named 'Desk Lamp'. Use a subquery. Present results sorted by LastName in ascending order and then FirstName in descending order.

Q. Show the LastName, FirstName, and Phone of all customers who have bought an Item named 'Desk Lamp'. Use a join, but do not use JOIN ON syntax. Present results sorted by LastName in ascending order and then FirstName in descending order.

R. Show the LastName, FirstName, and Phone of all customers who have bought an Item named 'Desk Lamp'. Use a join using JOIN ON syntax. Present results sorted by LastName in ascending order and then FirstName in descending order.

Figure 2-48
Sample Data for the QACS Database SALE_ITEM Table

SaleID	SaleItemID	ItemID	ItemPrice
1	1	1	$3,000.00
1	2	2	$500.00
2	1	3	$1,000.00
3	1	4	$50.00
4	1	5	$45.00
5	1	6	$250.00
6	1	7	$750.00
7	1	8	$250.00
8	1	9	$1,250.00
8	2	10	$1,750.00
9	1	11	$350.00
10	1	19	$5,000.00
10	2	21	$8,500.00
10	3	22	$750.00
11	1	17	$250.00
12	1	24	$50.00
13	1	20	$4,500.00
14	1	12	$3,200.00
14	2	14	$475.00
15	1	23	$800.00

S. Show the LastName, FirstName, and Phone of all customers who have bought an Item named 'Desk Lamp'. Use a combination of a join and a subquery. Present results sorted by LastName in ascending order and then FirstName in descending order.

T. Show the LastName, FirstName, and Phone of all customers who have bought an Item named 'Desk Lamp'. Use a combination of a join and a subquery that is different from the combination used for question S. Present results sorted by LastName in ascending order and then FirstName in descending order.

ItemID	ItemDescription	CompanyName	PurchaseDate	ItemCost	ItemPrice
1	Antique Desk	European Specialties	11/7/2012	$1,800.00	$3,000.00
2	Antique Desk Chair	Andrew Lee	11/10/2012	$300.00	$500.00
3	Dining Table Linens	Linens and Things	11/14/2012	$600.00	$1,000.00
4	Candles	Linens and Things	11/14/2012	$30.00	$50.00
5	Candles	Linens and Things	11/14/2012	$27.00	$45.00
6	Desk Lamp	Lamps and Lighting	11/14/2012	$150.00	$250.00
7	Dining Table Linens	Linens and Things	11/14/2012	$450.00	$750.00
8	Book Shelf	Denise Harrion	11/21/2012	$150.00	$250.00
9	Antique Chair	New York Brokerage	11/21/2012	$750.00	$1,250.00
10	Antique Chair	New York Brokerage	11/21/2012	$1,050.00	$1,750.00
11	Antique Candle Holder	European Specialties	11/28/2012	$210.00	$350.00
12	Antique Desk	European Specialties	1/5/2013	$1,920.00	$3,200.00
13	Antique Desk	European Specialties	1/5/2013	$2,100.00	$3,500.00
14	Antique Desk Chair	Specialty Antiques	1/6/2013	$285.00	$475.00
15	Antique Desk Chair	Specialty Antiques	1/6/2013	$339.00	$565.00
16	Desk Lamp	General Antiques	1/6/2013	$150.00	$250.00
17	Desk Lamp	General Antiques	1/6/2013	$150.00	$250.00
18	Desk Lamp	Lamps and Lighting	1/6/2013	$144.00	$240.00
19	Antique Dining Table	Denesha Walsh	1/10/2013	$3,000.00	$5,000.00
20	Antique Sideboard	Chris Bancroft	1/11/2013	$2,700.00	$4,500.00
21	Dining Table Chairs	Specialty Antiques	1/11/2013	$5,100.00	$8,500.00
22	Dining Table Linens	Linens and Things	1/12/2013	$450.00	$750.00
23	Dining Table Linens	Linens and Things	1/12/2013	$480.00	$800.00
24	Candles	Linens and Things	1/17/2013	$30.00	$50.00
25	Candles	Linens and Things	1/17/2013	$36.00	$60.00

Figure 2-49
QACS Database ITEM Table

Morgan Importing

James Morgan owns and operates Morgan Importing, which purchases antiques and home furnishings in Asia, ships those items to a warehouse facility in Los Angeles, and then sells these items in the United States. James tracks the Asian purchases and subsequent shipments of these items to Los Angeles by using a database to keep a list of items purchased, shipments of the purchased items, and the items in each shipment. His database includes the following tables:

ITEM (ItemID, Description, PurchaseDate, Store, City, Quantity, LocalCurrencyAmount, ExchangeRate)

SHIPMENT (ShipmentID, ShipperName, ShipperInvoiceNumber, DepartureDate, ArrivalDate, InsuredValue)

SHIPMENT_ITEM (*ShipmentID*, ShipmentItemID, *ItemID*, Value)

In the database schema above, the primary keys are underlined and the foreign keys are shown in italics. The database that James has created is named MI, and the three tables in the MI database schema are shown in Figure 2-50.

The column characteristics for the tables are shown in Figures 2-51, 2-52, and 2-53. The data for the tables are shown in Figures 2-54, 2-55, and 2-56. The relationship between ITEM

Figure 2-50
The MI Database

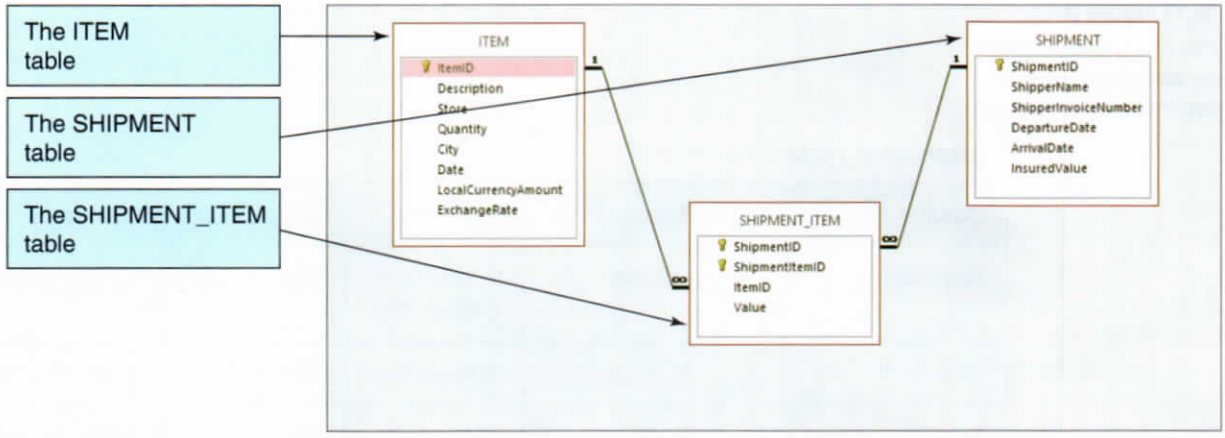


Figure 2-51
Column Characteristics for the MI Database ITEM Table

ITEM

Column Name	Type	Key	Required	Remarks
ItemID	AutoNumber	Primary Key	Yes	Surrogate Key
Description	Text (255)	No	Yes	Long Integer
PurchaseDate	Date	No	Yes	
Store	Text (50)	No	Yes	
City	Text (35)	No	Yes	
Quantity	Number	No	Yes	Long Integer
LocalCurrencyAmount	Number	No	Yes	Decimal, 18 Auto
ExchangeRate	Number	No	Yes	Decimal, 12 Auto

Figure 2-52
Column Characteristics for the MI Database SHIPMENT Table

SHIPMENT

Column Name	Type	Key	Required	Remarks
ShipmentID	AutoNumber	Primary Key	Yes	Surrogate Key
ShipperName	Text (35)	No	Yes	
ShipperInvoiceNumber	Number	No	Yes	Long Integer
DepartureDate	Date	No	No	
ArrivalDate	Date	No	No	
InsuredValue	Currency	No	No	Two Decimal Places

Figure 2-53
Column Characteristics for the MI Database SHIPMENT_ITEM Table

SHIPMENT_ITEM

Column Name	Type	Key	Required	Remarks
ShipmentID	Number	Primary Key, Foreign Key	Yes	Long Integer
ShipmentItemID	Number	Primary Key	Yes	Long Integer
ItemID	Number	Foreign Key	Yes	Long Integer
Value	Currency	No	Yes	Two Decimal Places

Figure 2-54
Sample Data for the MI Database ITEM Table

ItemID	Description	PurchaseDate	Store	City	Quantity	LocalCurrencyAmount	ExchangeRate
1	QE Dining Set	07-Apr-13	Eastern Treasures	Manila	2	403405	0.01774
2	Willow Serving Dishes	15-Jul-13	Jade Antiques	Singapore	75	102	0.5903
3	Large Bureau	17-Jul-13	Eastern Sales	Singapore	8	2000	0.5903
4	Brass Lamps	20-Jul-13	Jade Antiques	Singapore	40	50	0.5903

ShipmentID	ShipperName	ShipperInvoiceNumber	DepartureDate	ArrivalDate	InsuredValue
1	ABC Trans-Oceanic	2008651	10-Dec-12	15-Mar-13	$15,000.00
2	ABC Trans-Oceanic	2009012	10-Jan-13	20-Mar-13	$12,000.00
3	Worldwide	49100300	05-May-13	17-Jun-13	$20,000.00
4	International	399400	02-Jun-13	17-Jul-13	$17,500.00
5	Worldwide	84899440	10-Jul-13	28-Jul-13	$25,000.00
6	International	488955	05-Aug-13	11-Sep-13	$18,000.00

Figure 2-55
Sample Data for the MI Database SHIPMENT Table

ShipmentID	ShipmentItemID	ItemID	Value
3	1	1	$15,000.00
4	1	4	$1,200.00
4	2	3	$9,500.00
4	3	2	$4,500.00

Figure 2-56
Sample Data for the MI Database SHIPMENT_ITEM Table

and SHIPMENT_ITEM should enforce referential integrity, and although it should cascade updates, it should not cascade deletions. The relationship between SHIPMENT and SHIPMENT_ITEM should enforce referential integrity and cascade both updates and deletions.

We recommend that you create a Microsoft Access 2013 database named *MI-CH02.accdb* using the database schema, column characteristics, and data shown above and then use this database to test your solutions to the questions in this section. Alternatively, SQL scripts for creating the *MI-CH02* database in Microsoft SQL Server, Oracle Database, and MySQL are available on our Web site at *www.pearsonhighered.com/kroenke*.

Write SQL statements and show the results based on the MI data for each of the following:

A. Show all data in each of the tables.

B. List the ShipmentID, ShipperName, and ShipperInvoiceNumber of all shipments.

C. List the ShipmentID, ShipperName, and ShipperInvoiceNumber for all shipments that have an insured value greater than $10,000.00.

D. List the ShipmentID, ShipperName, and ShipperInvoiceNumber of all shippers whose name starts with 'AB'.

E. Assume DepartureDate and ArrivalDate are in the format MM/DD/YY. List the ShipmentID, ShipperName, ShipperInvoiceNumber, and ArrivalDate of all shipments that departed in December.

F. Assume DepartureDate and ArrivalDate are in the format MM/DD/YY. List the ShipmentID, ShipperName, ShipperInvoiceNumber, and ArrivalDate of all shipments that departed on the tenth day of any month.

G. Determine the maximum and minimum InsuredValue.

H. Determine the average InsuredValue.

I. Count the number of shipments.

J. Show ItemID, Description, Store, and a calculated column named USCurrencyAmount that is equal to LocalCurrencyAmount multiplied by the ExchangeRate for all rows of ITEM.

K. Group item purchases by City and Store.

L. Count the number of purchases having each combination of City and Store.

M. Show the ShipperName, ShipmentID and DepartureDate of all shipments that have an item with a value of $1,000.00 or more. Use a subquery. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

N. Show the ShipperName, ShipmentID, and DepartureDate of all shipments that have an item with a value of $1,000.00 or more. Use a join. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

O. Show the ShipperName, ShipmentID, and DepartureDate of the shipment for items that were purchased in Singapore. Use a subquery. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

P. Show the ShipperName, ShipmentID, and DepartureDate of all shipments that have an item that was purchased in Singapore. Use a join, but do not use JOIN ON syntax. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

Q. Show the ShipperName, ShipmentID, and DepartureDate of all shipments that have an item that was purchased in Singapore. Use a join using JOIN ON syntax. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

R. Show the ShipperName, ShipmentID, the DepartureDate of the shipment, and Value for items that were purchased in Singapore. Use a combination of a join and a subquery. Present results sorted by ShipperName in ascending order and then DepartureDate in descending order.

S. Show the ShipperName, ShipmentID, the DepartureDate of the shipment, and Value for items that were purchased in Singapore. Also show the ShipperName, ShipmentID, and DepartureDate for all other shipments. Present results sorted by Value in ascending order, then ShipperName in ascending order, and then DepartureDate in descending order.