4

Creating a Simple Query

"Think like a wise man but communicate
in the language of the people."
—William Butler Yeats

Topics Covered in This Chapter

Introducing SELECT

The SELECT Statement

A Quick Aside: Data versus Information

Translating Your Request into SQL

Eliminating Duplicate Rows

Sorting Information

Saving Your Work

Sample Statements

Summary

Problems for You to Solve

Now that you've learned a little bit about the history of SQL, it's time to jump right in and learn the language itself. As we mentioned in the Introduction, we're going to spend most of this book covering the data manipulation portion of the language. So our initial focus will be on the true workhorse of SQL—the SELECT statement.

Introducing SELECT

Above all other keywords, SELECT truly lies at the heart of SQL. It is the cornerstone of the most powerful and complex statement within the language and the means by which you retrieve information from the tables in your database. You use SELECT in conjunction with other keywords and clauses to find and view information in an almost limitless number of ways. Nearly any question regarding who, what, where, when, or even what if and how many can be answered with SELECT. As long as you've designed your database properly and collected the appropriate data, you can get the answers you need to make sound decisions for your organization. As you'll discover when you get to Part V, Modifying Sets of Data, you'll apply many of the techniques you learn about SELECT to create UPDATE, INSERT, and DELETE statements.

The SELECT operation in SQL can be broken down into three smaller operations, which we will refer to as the SELECT statement, the SELECT expression, and the SELECT query. (Breaking down the SELECT operation in this manner will make it far easier to understand and to appreciate its complexity.) Each of these operations provides its own set of keywords and clauses, providing you with the flexibility to create a final SQL statement that is appropriate for the question you want to pose to the database. As you'll learn in later chapters, you can even combine the operations in various ways to answer very complex questions.

In this chapter, we'll begin our discussion of the SELECT statement and take a brief look at the SELECT query. We'll then examine the SELECT statement in more detail as we work through to Chapter 5, Getting More Than Simple Columns, and Chapter 6, Filtering Your Data.

❖ **Note** In other books about relational databases, you'll sometimes see he word *relation* used for *table*, and you might encounter *tuple* or *record* for *row* and perhaps *attribute* or *field* for *column*. However, the SQL Standard specifically uses the terms *table*, *row*, and *column* to refer to these particular elements of a database structure. We'll stay consistent with the SQL Standard and use these latter three terms throughout the remainder of the book.

The SELECT Statement

The SELECT statement forms the basis of every question you pose to the database. When you create and execute a SELECT statement, you are querying the database. (We know it sounds a little obvious, but we want to make certain that everyone reading this starts from the same point of reference.) In fact, many RDBMS programs allow you to save a SELECT statement as a *query*, *view*, *function*, or *stored procedure*. Whenever someone says she is going to query the database, you know that she's going to execute some sort of SELECT statement. Depending on the RDBMS program, SELECT statements can be executed directly from a command line window, from an interactive Query by Example (QBE) grid, or from within a block of programming code. Regardless of how you choose to define and execute it, the syntax of the SELECT statement is always the same.

❖ **Note** Many database systems provide extensions to the SQL Standard to allow you to build complex programming statements (such as If . . . Then . . . Else) in functions and stored procedures, but the specific syntax is unique to each different product. It is far beyond the scope of this book to cover even one or two of these programming languages—such as Microsoft SQL Server's Transact-SQL or Oracle's PL/SQL. You'll still use the cornerstone SELECT statement when you build functions and stored procedures for your particular database system. Throughout this book, we'll use the term *view* to refer to a saved SQL statement even though you might embed your SQL statement in a function or procedure.

A SELECT statement is composed of several distinct keywords, known as *clauses*. You define a SELECT statement by using various configurations of these clauses to retrieve the information you require. Some of these clauses are required, although others are optional. Additionally, each clause has one or more keywords that represent required or optional values. These values are used by the clause to help retrieve the information requested by the SELECT statement as a whole. Figure 4–1 (on page 73) shows a diagram of the SELECT statement and its clauses.

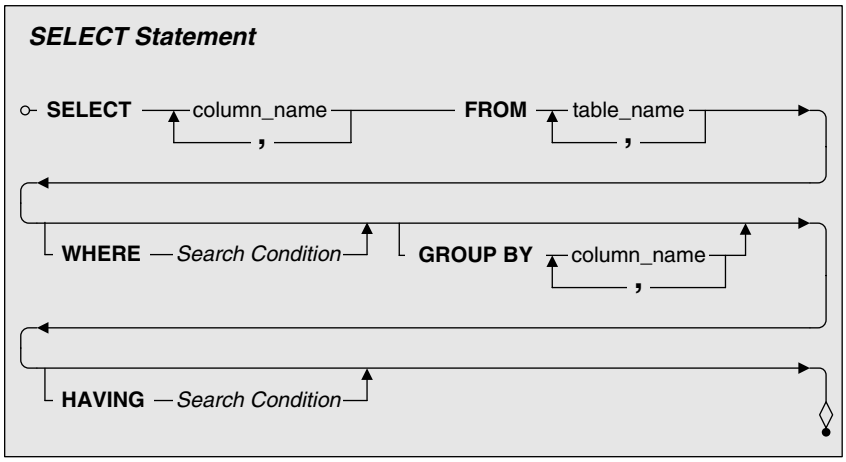

Figure 4–1 *A diagram of the SELECT statement*

❖ **Note** The syntax diagram in Figure 4–1 reflects a rudimentary SELECT statement. We'll continue to update and modify the diagram as we introduce and work with new keywords and clauses. So for those of you who might have some previous experience with SQL statements, just be patient and bear with us for the time being.

Here's a brief summary of the clauses in a SELECT statement.

- **SELECT**—This is the primary clause of the SELECT statement and is absolutely required. You use it to specify the columns you want in the result set of your query. The columns themselves are drawn from the table or view you specify in the FROM clause. (You can also draw them from several tables simultaneously, but we'll discuss this later in Part III, Working with Multiple Tables.) You can also use aggregate functions, such as Sum(HoursWorked), or mathematical expressions, such as Quantity * Price, in this clause.
- **FROM**—This is the second most important clause in the SELECT statement and is also required. You use the FROM clause to specify the tables or views from which to draw the columns you've listed in the SELECT clause. You can use this clause in more complex ways, but we'll discuss this in later chapters.

- **WHERE**—This is an optional clause that you use to filter the rows returned by the FROM clause. The WHERE keyword is followed by an expression, technically known as a *predicate*, that evaluates to true, false, or unknown. You can test the expression by using standard comparison operators, Boolean operators, or special operators. We'll discuss all the elements of the WHERE clause in Chapter 6.
- **GROUP BY**—When you use aggregate functions in the SELECT clause to produce summary information, you use the GROUP BY clause to divide the information into distinct groups. Your database system uses any column or list of columns following the GROUP BY keywords as grouping columns. The GROUP BY clause is optional, and we'll examine it further in Chapter 13, Grouping Data.
- **HAVING**—The HAVING clause filters the result of aggregate functions in grouped information. It is similar to the WHERE clause in that the HAVING keyword is followed by an expression that evaluates to true, false, or unknown. You can test the expression by using standard comparison operators, Boolean operators, or special operators. HAVING is also an optional clause, and we'll take a closer look at it in Chapter 14, Filtering Grouped Data.

We're going to work with a very basic SELECT statement at first, so we'll focus on the SELECT and FROM clauses. We'll add the other clauses, one by one, as we work through the other chapters to build more complex SELECT statements.

A Quick Aside: Data versus Information

Before we pose the first query to the database, one thing must be perfectly clear: There is a distinct difference between *data* and *information*. In essence, data is what you store in the database, and information is what you retrieve from the database. This distinction is important for you to understand because it helps you to keep things in proper perspective. Remember that a database is designed to provide meaningful information to someone within your organization. However, the information can be provided only if the appropriate data exists in the database and if the database itself has been structured in such a way to support that information. Let's examine these terms in more detail.

The values that you store in the database are data. Data is static in the sense that it remains in the same state until you modify it by some manual or automated process. Figure 4–2 shows some sample data.

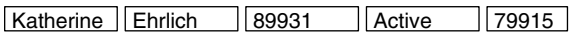

Figure 4–2 *An example of basic data*

On the surface, this data is meaningless. For example, there is no easy way for you to determine what 89931 represents. Is it a ZIP Code? Is it a part number? Even if you know it represents a customer identification number, is it associated with Katherine Ehrlich? There's no way to know until the data is processed. After you process the data so that it is meaningful and useful when you work with it or view it, the data becomes information. Information is dynamic in that it constantly changes relative to the data stored in the database and also in its ability to be processed and presented in an unlimited number of ways. You can show information as the result of a SELECT statement, display it in a form on your computer screen, or print it on paper as a report. But the point to remember is that you must process your data in a manner that enables you to turn it into meaningful information.

Figure 4–3 shows the data from the previous example transformed into information on a customer screen. This illustrates how the data can be manipulated in such a way that it is now meaningful to anyone who views it.

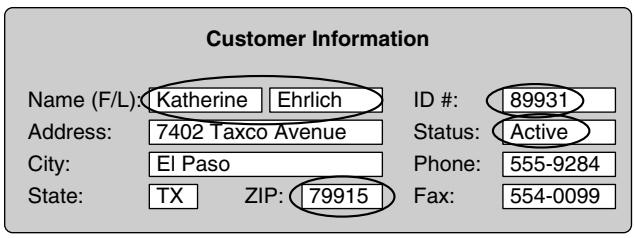

Figure 4–3 *An example of data processed into information*

When you work with a SELECT statement, you use its clauses to manipulate *data*, but the statement itself returns *information*. Get the picture?

There's one last issue we need to address. When you execute a SELECT statement, it usually retrieves one or more rows of information—the exact

number depends on how you construct the statement. These rows are collectively known as a *result set*, which is the term we use throughout the remainder of the book. This name makes perfect sense because you always work with sets of data whenever you use a relational database. (Remember that the relational model is based, in part, on set theory.) You can easily view the information in a result set and, in many cases, you can modify its data. But, once again, it all depends on how you construct your SELECT statement.

So let's get down to business and start using the SELECT statement.

Translating Your Request into SQL

When you request information from the database, it's usually in the form of a question or a statement that implies a question. For example, you might formulate statements such as these:

"Which cities do our customers live in?"

"Show me a current list of our employees and their phone numbers."

"What kind of classes do we currently offer?"

"Give me the names of the folks on our staff and the dates they were hired."

After you know what you want to ask, you can translate your request into a more formal statement. You compose the translation using this form:

Select <item> from the <source>

Start by looking at your request and replacing words or phrases such as *"list*," *"show me*," *"what*," *"which*," and *"who"* with the word "*Select*." Next, identify any nouns in your request, and determine whether a given noun represents an item you want to see or the name of a table in which an item might be stored. If it's an item, use it as a replacement for <item> in the translation statement. If it's a table name, use it as a replacement for <source>. If you translate the first question listed earlier, your statement looks something like this:

Select city from the customers table

After you define your translation statement, you need to turn it into a full-fledged SELECT statement using the SQL syntax shown in Figure 4–4. The

first step, however, is to clean up your translation statement. You do so by crossing out any word that is not a noun representing the name of a column or table or that is not a word specifically used in the SQL syntax. Here's how the translation statement looks during the process of cleaning it up:

Select city from ~~the~~ customers ~~table~~

Remove the words you've crossed out, and you now have a complete SELECT statement.

```
SELECT City FROM Customers
```

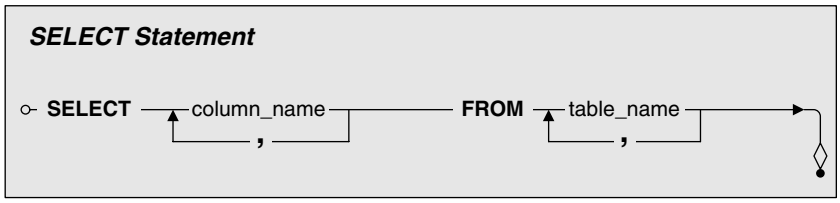

Figure 4–4 *The syntax of a simple SELECT statement*

You can use the three-step technique we just presented on any request you send to your database. In fact, we use this technique throughout most of the book, and we encourage you to use it while you're beginning to learn how to build these statements. However, you'll eventually merge these steps into one seamless operation as you get more accustomed to writing SELECT statements.

Remember that you'll work mostly with columns and tables when you're beginning to learn how to use SQL. The syntax diagram in Figure 4–4 reflects this fact by using column_name in the SELECT clause and table_name in the FROM clause. In the next chapter, you'll learn how to use other terms in these clauses to create more complex SELECT statements.

You probably noticed that the request we used in the previous example is relatively straightforward. It was easy to both redefine it as a translation statement and identify the column names that were present in the statement. But what if a request is not as straightforward and easy to translate, and it's difficult to identify the columns you need for the SELECT clause? The easiest course of action is to refine your request and make it more specific. For example, you can refine a request such as *"Show me the information on our clients"* by recasting it more clearly as *"List the name, city, and phone number for each*

of our clients." If refining the request doesn't solve the problem, you still have two other options. Your first alternative is to determine whether the table specified in the FROM clause of the SELECT statement contains any column names that can help to clarify the request and thus make it easier to define a translation statement. Your second alternative is to examine the request more closely and determine whether a word or phrase it contains *implies* any column names. Whether you can use either or both alternatives depends on the request itself. Just remember that you do have techniques available when you find it difficult to define a translation statement. Let's look at an example of each technique and how you can apply it in a typical scenario.

To illustrate the first technique, let's say you're trying to translate the following request.

> *"I need the names and addresses of all our employees."*

This looks like a straightforward request on the surface. But if you review this request again, you'll find one minor problem: Although you can determine the table you need (Employees) for the translation statement, there's nothing within the request that helps you identify the specific columns you need for the SELECT clause. Although the words *"names"* and *"addresses"* appear in the request, they are terms that are general in nature. You can solve this problem by reviewing the table you identified in the request and determining whether it contains any columns you can substitute for these terms. If so, use the column names in the translation statement. (You can opt to use generic versions of the column names in the translation statement if it will help you visualize the statement more clearly. However, you will need to use the actual column names in the SQL syntax.) In this case, look for column names in the Employees table shown in Figure 4–5 that could be used in place of the words "names" and "addresses."

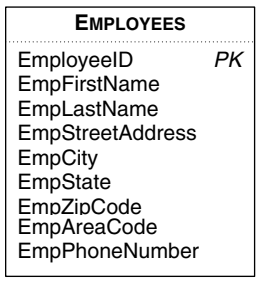

Figure 4–5 *The structure of the Employees table*

To fully satisfy the need for "names" and "addresses," you will indeed use six columns from this table. EmpFirstName and EmpLastName will both replace "names" in the request, and EmpStreetAddress, EmpCity, EmpState, and EmpZipCode will replace "addresses." Now, apply the entire translation process to the request, which we've repeated for your convenience. (We'll use generic forms of the column names for the translation statement and the actual column names in the SQL syntax.)

"I need the names and addresses of all our employees."

Translation	Select first name, last name, street address, city, state, and ZIP Code from the employees table
Clean Up	Select first name, last name, street address, city, state, ~~and~~ ZIP Code from ~~the~~ employees ~~table~~
SQL	`SELECT EmpFirstName, EmpLastName, EmpStreetAddress, EmpCity, EmpState, EmpZipCode FROM Employees`

❖ **Note** This example clearly illustrates how to use multiple columns in a SELECT clause. We'll discuss this technique in more detail later in this section.

The next example illustrates the second technique, which involves searching for implied columns within the request. Let's assume you're trying to put the following request through the translation process.

"What kind of classes do we currently offer?"

At first glance, it might seem difficult to define a translation statement from this request. The request doesn't indicate any column names, and without even one item to select, you can't create a complete translation statement. What do you do now? Take a closer look at each word in the request and determine whether there is one that *implies* a column name within the Classes table. Before you read any further, take a moment to study the request again. Can you find such a word?

In this case, the word "kind" might imply a column name in the Classes table. Why? Because a kind of class can also be thought of as a category of class. If there is a category column in the Classes table, then you have the column

name you need to complete the translation statement and, by inference, the SELECT statement. Let's assume that there is a category column in the Classes table and take the request through the three-step process once again.

"What kind of classes do we currently offer?"

Translation	Select category from the classes table
Clean Up	Select category from ~~the~~ classes ~~table~~
SQL	`SELECT Category` `FROM Classes`

As the example shows, this technique involves using synonyms as replacements for certain words or phrases within the request. If you identify a word or phrase that might imply a column name, try to replace it with a synonym. The synonym you choose might indeed identify a column that exists in the database. However, if the first synonym that comes to mind doesn't work, try another. Continue this process until you either find a synonym that does identify a column name or until you're satisfied that neither the original word nor any of its synonyms represent a column name.

❖ **Note** Unless we indicate otherwise, all column names and table names used in the SQL syntax portion of the examples are drawn from the sample databases in Appendix B, Schema for the Sample Databases. This convention applies to all examples for the remainder of the book.

Expanding the Field of Vision

You can retrieve multiple columns within a SELECT statement as easily as you can retrieve a single column. List the names of the columns you want to use in the SELECT clause, and separate each name in the list with a comma. In the syntax diagram shown in Figure 4–6, the option to use more than one column is indicated by a line that flows from right to left beneath column_name. The comma in the middle of the line denotes that you must insert a comma before the next column name you want to use in the SELECT clause.

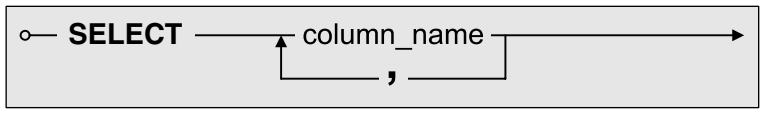

Figure 4–6 *The syntax for using multiple columns in a SELECT clause*

The option to use multiple columns in the SELECT statement provides you with the means to answer questions such as these.

"Show me a current list of our employees and their phone numbers."

Translation	Select the last name, first name, and phone number of all our employees from the employees table
Clean Up	Select ~~the~~ last name, first name, ~~and~~ phone number ~~of all our employees~~ from ~~the~~ employees ~~table~~
SQL	`SELECT EmpLastName, EmpFirstName, EmpPhoneNumber FROM Employees`

"What are the names and prices of the products we carry, and under what category is each item listed?"

Translation	Select the name, price, and category of every product from the products table
Clean Up	Select ~~the~~ name, price, ~~and~~ category ~~of every product~~ from ~~the~~ products ~~table~~
SQL	`SELECT ProductName, RetailPrice, Category FROM Products`

You gain the advantage of seeing a wider spectrum of information when you work with several columns in a SELECT statement. Incidentally, the sequence of the columns in your SELECT clause is not important—you can list the columns in any order you want. This gives you the flexibility to view the same information in a variety of ways.

For example, let's say you're working with the table shown in Figure 4–7, and you're asked to pose the following request to the database.

"Show me a list of subjects, the category each belongs to, and the code we use in our catalog. But I'd like to see the name first, followed by the category and then the code."

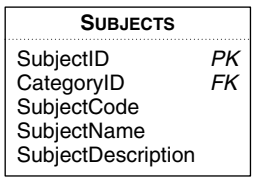

Figure 4–7 *The structure of the Subjects table*

You can still transform this request into an appropriate SELECT statement, even though the person making the request wants to see the columns in a specific order. Just list the column names in the order specified when you define the translation statement. Here's how the process looks when you transform this request into a SELECT statement.

Translation	Select the subject name, category ID, and subject code from the subjects table
Clean Up	Select ~~the~~ subject name, category ID, ~~and~~ subject code from ~~the~~ subjects ~~table~~
SQL	`SELECT SubjectName, CategoryID, SubjectCode` `FROM Subjects`

Using a Shortcut to Request All Columns

There is no limit to the number of columns you can specify in the SELECT clause—in fact, you can list all the columns from the source table. The following example shows the SELECT statement you use to specify all the columns from the Subjects table in Figure 4–7.

```
SQL         SELECT SubjectID, CategoryID, SubjectCode,
               SubjectName, SubjectDescription
            FROM Subjects
```

When you specify all the columns from the source table, you'll have a lot of typing to do if the table contains a number of columns! Fortunately, the SQL Standard specifies the asterisk as a shortcut you can use to shorten the statement considerably. The syntax diagram in Figure 4–8 shows that you can use the asterisk as an alternative to a list of columns in the SELECT clause.

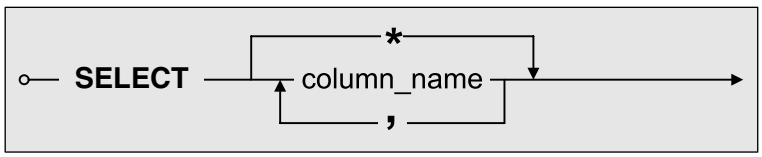

Figure 4–8 *The syntax for the asterisk shortcut*

Place the asterisk immediately after the SELECT clause when you want to specify all the columns from the source table in the FROM clause. For

example, here's how the preceding SELECT statement looks when you use the shortcut.

```
SQL         SELECT *
            FROM Subjects
```

You'll certainly do less typing with this statement! However, one issue arises when you create SELECT statements in this manner: The asterisk represents all of the columns that *currently exist* in the source table, and adding or deleting columns affects what you see in the result set of the SELECT statement. (Oddly enough, the SQL Standard states that adding or deleting columns *should not* affect your result set.) This issue is important only if you must see the same columns in the result set consistently. Your database system will not warn you if columns have been deleted from the source table when you use the asterisk in the SELECT clause, but it will raise a warning when it can't find a column you *explicitly* specified. Although this does not pose a real problem for our purposes, it will be an important issue when you delve into the world of programming with SQL. Our rule of thumb is this: Use the asterisk only when you need to create a "quick and dirty" query to see all the information in a given table. Otherwise, specify all the columns you need for the query. In the end, the query will return exactly the information you need and will be more self-documenting.

The examples we've seen so far are based on simple requests that require columns from only one table. You'll learn how to work with more complex requests that require columns from several tables in Part III.

Eliminating Duplicate Rows

When working with SELECT statements, you'll inevitably come across result sets with duplicate rows. There is no cause for alarm if you see such a result set. Use the DISTINCT keyword in your SELECT statement, and the result set will be free and clear of all duplicate rows. Figure 4–9 shows the syntax diagram for the DISTINCT keyword.

As the diagram illustrates, DISTINCT is an optional keyword that precedes the list of columns specified in the SELECT clause. The DISTINCT keyword asks your database system to evaluate the values of all the columns *as a single unit* on a row-by-row basis and eliminate any redundant rows it finds. The remaining unique rows are then returned to the result set. The following

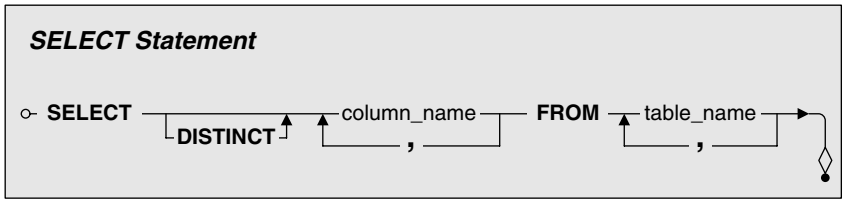

Figure 4–9 *The syntax for the DISTINCT keyword*

example shows what a difference the DISTINCT keyword can make under the appropriate circumstances.

Let's say you're posing the following request to the database.

"Which cities are represented by our bowling league membership?"

The question seems easy enough, so you take it through the translation process.

Translation	Select city from the bowlers table
Clean Up	Select city from ~~the~~ bowlers ~~table~~
SQL	`SELECT City` `FROM Bowlers`

The problem is that the result set for this SELECT statement shows *every occurrence* of each city name found in the Bowlers table. For example, if there are 20 people from Bellevue and 7 people from Kent and 14 people from Seattle, the result set displays 20 occurrences of Bellevue, 7 occurrences of Kent, and 14 occurrences of Seattle. Clearly, this redundant information is unnecessary. All you want to see is a *single* occurrence of each city name found in the Bowlers table. You resolve this problem by using the DISTINCT keyword in the SELECT statement to eliminate the redundant information.

Let's run the request through the translation process once again using the DISTINCT keyword. Note that we now include the word "distinct" in both the Translation step and the Clean Up step.

"Which cities are represented by our bowling league membership?"

Translation	Select the distinct city values from the bowlers table
Clean Up	Select ~~the~~ distinct city ~~values~~ from ~~the~~ bowlers ~~table~~
SQL	`SELECT DISTINCT City` `FROM Bowlers`

The result set for this SELECT statement displays exactly what you're looking for—a single occurrence of each distinct (or unique) city found in the Bowlers table.

You can use the DISTINCT keyword on multiple columns as well. Let's modify the previous example by requesting both the city and the state from the Bowlers table. Our new SELECT statement looks like this.

```
SELECT DISTINCT City, State FROM Bowlers
```

This SELECT statement returns a result set that contains unique records and shows definite distinctions between cities with the same name. For example, it shows the distinction between "Portland, ME," "Portland, OR," "Hollywood, CA," and "Hollywood, FL." It's worthwhile to note that most database systems sort the output in the sequence in which you specify the columns, so you'll see these values in the sequence "Hollywood, CA," "Hollywood, FL," "Portland, ME," and "Portland, OR." However, the SQL Standard does not require the result to be sorted in this order. If you want to guarantee the sort sequence, read on to the next section to learn about the ORDER BY clause.

The DISTINCT keyword is a very useful tool under the right circumstances. Use it only when you really want to see unique rows in your result set.

❖ **Caution** For database systems that include a graphical interface, you can usually request that the result set of a query be displayed in an updatable grid of rows and columns. You can type a new value in a column on a row, and the database system updates the value stored in your table. (Your database system actually executes an UPDATE query on your behalf behind the scenes—you can read more about that in Chapter 15, Updating Sets of Data.)

However, in all database systems that we studied, when you include the DISTINCT keyword, the resulting set of rows cannot be updated. To be able to update a column in a row, your database system needs to be able to uniquely identify the specific row and column you want to change. When you use DISTINCT, the values you see in each row are the result of evaluating perhaps dozens of duplicate rows. If you try to update one of the columns, your database won't know which specific row to change. Your database system also doesn't know if perhaps you mean to change all the rows with the same duplicate value.

Sorting Information

At the beginning of this chapter, we said that the SELECT operation can be broken down into three smaller operations: the SELECT statement, the SELECT expression, and the SELECT query. We also stated that you can combine these operations in various ways to answer complex requests. However, you also need to combine these operations in order to sort the rows of a result set.

By definition, the rows of a result set returned by a SELECT statement are unordered. The sequence in which they appear is typically based on their physical position in the table. (The actual sequence is often determined dynamically by your database system based on how it decides to most efficiently satisfy your request.) The only way to sort the result set is to embed the SELECT statement within a SELECT query, as shown in Figure 4–10. We define a SELECT query as a SELECT statement with an ORDER BY clause. The ORDER BY clause of the SELECT query lets you specify the sequence of rows in the final result set. As you'll learn in later chapters, you can actually embed a SELECT statement within another SELECT statement or SELECT expression to answer very complex questions. However, the SELECT query cannot be embedded at any level.

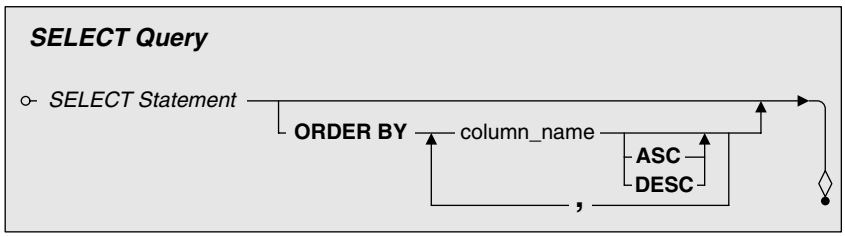

Figure 4–10 *The syntax diagram for the SELECT query*

❖ **Note** Throughout this book, we use the same terms you'll find in the SQL Standard or in common usage in most database systems. The 2003 SQL Standard, however, defines the ORDER BY clause as part of a *cursor* (an object that you define inside an application program), as part of an *array* (a list of values that form a logical table such as a *subquery,* discussed in Chapter 11, Subqueries), or as part of a *scalar subquery* (a subquery that returns only one value). A complete discussion of cursors and arrays is beyond the scope of this book. Because nearly all implementations of SQL allow you to include an ORDER BY clause at the end of a SELECT statement that you can save in a view, we invented the term *SELECT query* to describe this type of statement. This also allows us to discuss the concept of sorting the final output of a query for display online or for use in a report. It's our understanding that the draft 2007/2008 standard does allow using ORDER BY in more places, but we'll use this separate construct in this book to cover the topic.

The ORDER BY clause allows you to sort the result set of the specified SELECT statement by one or more columns and also provides the option of specifying an ascending or descending sort order for each column. The only columns you can use in the ORDER BY clause are those that are currently listed in the SELECT clause. (Although this requirement is specified in the SQL Standard, some vendor implementations allow you to disregard it completely. However, we comply with this requirement in all the examples used throughout the book.) When you use two or more columns in an ORDER BY clause, separate each column with a comma. The SELECT query returns a final result set once the sort is complete.

❖ **Note** The ORDER BY clause *does not* affect the physical order of the rows in a table. If you do need to change the physical order of the rows, refer to your database software's documentation for the proper procedure.

First Things First: Collating Sequences

Before we look at some examples using the SELECT query, a brief word on collating sequences is in order. The manner in which the ORDER BY clause sorts the information depends on the collating sequence used by your database software. The collating sequence determines the order of precedence for every character listed in the current language character set specified by

your operating system. For example, it identifies whether lowercase letters will be sorted before uppercase letters, or whether case will even matter. Check your database software's documentation, and perhaps consult your database administrator to determine the default collating sequence for your database. For more information on collating sequences, see the subsection Comparing String Values: A Caution in Chapter 6.

Let's Now Come to Order

With the availability of the ORDER BY clause, you can present the information you retrieve from the database in a more meaningful fashion. This applies to simple requests as well as complex ones. You can now rephrase your requests so that they also indicate sorting requirements. For example, a question such as *"What are the categories of classes we currently offer?"* can be restated as *"List the categories of classes we offer and show them in alphabetical order."*

Before beginning to work with the SELECT query, you need to adjust the way you define a translation statement. This involves adding a new section at the end of the translation statement to account for the new sorting requirements specified within the request. Use this new form to define the translation statement.

Select <item> from the <source> **and order by <column(s)>**

Now that your request will include phrases such as "sort the results by city," "show them in order by year," or "list them by last name and first name," study the request closely to determine which column or columns you need to use for sorting purposes. This is a simple exercise because most people use these types of phrases, and the columns needed for the sort are usually self-evident. After you identify the appropriate column or columns, use them as a replacement for <column(s)> in the translation statement. Let's take a look at a simple request to see how this works.

"List the categories of classes we offer and show them in alphabetical order."

Translation	Select category from the classes table and order by category
Clean Up	Select category from ~~the~~ classes ~~table and~~ order by category

SQL	`SELECT Category`
	`FROM Classes`
	`ORDER BY Category`

In this example, you can assume that Category will be used for the sort because it's the only column indicated in the request. You can also assume that the sort should be in ascending order because there's nothing in the request to indicate the contrary. This is a safe assumption. According to the SQL Standard, ascending order is automatically assumed if you don't specify a sort order. However, if you want to be absolutely explicit, insert ASC after Category in the ORDER BY clause.

In the following request, the column needed for the sort is more clearly defined.

"Show me a list of vendor names in ZIP Code order."

Translation	Select vendor name and ZIP Code from the vendors table and order by ZIP Code
Clean Up	Select vendor name ~~and~~ ZIP Code from ~~the~~ vendors ~~table and~~ order by ZIP Code

SQL	`SELECT VendName, VendZipCode`
	`FROM Vendors`
	`ORDER BY VendZipCode`

In general, most people will tell you if they want to see their information in descending order. When this situation arises and you need to display the result set in reverse order, insert the DESC keyword after the appropriate column in the ORDER BY clause. For example, here's how you would modify the SELECT statement in the previous example when you want to see the information sorted by ZIP Code in descending order.

```
SQL         SELECT VendName, VendZipCode
            FROM Vendors
            ORDER BY VendZipCode DESC
```

The next example illustrates a more complex request that requires a multicolumn sort. The only difference between this example and the previous two examples is that this example uses more columns in the ORDER BY clause. Note that the columns are separated with commas, which is in accordance with the syntax diagram shown in Figure 4–10.

"Display the names of our employees, including their phone number and ID number, and list them by last name and first name."

Translation	Select last name, first name, phone number, and employee ID from the employees table and order by last name and first name
Clean Up	Select last name, first name, phone number, ~~and~~ employee ID from ~~the~~ employees ~~table and~~ order by last name ~~and~~ first name

SQL

```
SELECT EmpLastName, EmpFirstName,
   EmpPhoneNumber, EmployeeID
FROM Employees
ORDER BY EmpLastName, EmpFirstName
```

One of the interesting things you can do with the columns in an ORDER BY clause is to specify a different sort order for each column. In the previous example, you can specify a descending sort for the column containing the last name and an ascending sort for the column containing the first name. Here's how the SELECT statement looks when you make the appropriate modifications.

SQL

```
SELECT EmpLastName, EmpFirstName, EmpPhoneNumber,
   EmployeeID
FROM Employees
ORDER BY EmpLastName DESC, EmpFirstName ASC
```

Although you don't need to use the ASC keyword explicitly, the statement is more self-documenting if you include it.

The previous example brings an interesting question to mind: Is any importance placed on the sequence of the columns in the ORDER BY clause? The answer is *"Yes!"* The sequence is important because your database system will evaluate the columns in the ORDER BY clause from left to right. Also, the importance of the sequence grows in direct proportion to the number of columns you use. Always sequence the columns in the ORDER BY clause properly so that the result sorts in the appropriate order.

❖ **Note** The database products from Microsoft (Microsoft Office Access and Microsoft SQL Server) include an interesting extension that allows you to request a subset of rows based on your ORDER BY clause by using the TOP keyword in the SELECT clause. For example, you can find out the five most expensive products in the Sales Orders database by requesting:

```
SELECT TOP 5 ProductName, RetailPrice
   FROM Products
   ORDER BY RetailPrice DESC
```

The database sorts all the rows from the Products table descending by price and then returns the top five rows. Both database systems also allow you to specify the number of rows returned as a percentage of all the rows. For example, you can find out the top 10 percent of products by price by requesting:

```
SELECT TOP 10 PERCENT ProductName, RetailPrice
   FROM Products
   ORDER BY RetailPrice DESC
```

In fact, if you want to specify ORDER BY in a view, SQL Server requires that you include the TOP keyword. If you want all rows, you must specify TOP 100 PERCENT. For this reason, you'll see that all the sample views in SQL Server that include an ORDER BY clause also specify TOP 100 PERCENT. There is no such restriction in Microsoft Access.

Saving Your Work

Save your SELECT statements—every major database software program provides a way for you to save them! Saving your statements eliminates the need to recreate them every time you want to make the same request to the database. When you save your SELECT statement, assign a meaningful name that will help you remember what type of information the statement provides. And if your database software allows you to do so, write a concise description of the statement's purpose. The value of the description will become quite clear when you haven't seen a particular SELECT statement for some time and you need to remember why you constructed it in the first place.

A saved SELECT statement is categorized as a query in some database programs and as a view, function, or stored procedure in others. Regardless of its

designation, every database program provides you with a means to execute, or run, the saved statement and work with its result set.

❖ **Note** For the remainder of this discussion, we'll use the word *query* to represent the saved SELECT statement and *execute* to represent the method used to work with it.

Two common methods are used to execute a query. The first is an interactive device (such as a command on a toolbar or query grid), and the second is a block of programming code. You'll use the first method quite extensively. There's no need to worry about the second method until you begin working with your database software's programming language. Although it's our job to teach you how to create and use SQL statements, it's your job to learn how to create, save, and execute them in your database software program.

Sample Statements

Now that we've covered the basic characteristics of the SELECT statement and SELECT query, let's take a look at some examples of how these operations are applied in different scenarios. These examples encompass each of the sample databases, and they illustrate the use of the SELECT statement, the SELECT query, and the two supplemental techniques used to establish columns for the translation statement. We've also included sample result sets that would be returned by these operations and placed them immediately after the SQL syntax line. The name that appears immediately above a result set has a twofold purpose: It identifies the result set itself, and it is also the name that we assigned to the SQL statement in the example.

In case you're wondering why we assigned a name to each SQL statement, it's because we saved them! In fact, we've named and saved all the SQL statements that appear in the examples here and throughout the remainder of the book. Each is stored in the appropriate sample database (as indicated within the example), and we prefixed the names of the queries relevant to this chapter with "CH04." You can follow the instructions in the Introduction of this book to load the samples onto your computer. This gives you the opportunity to see these statements in action before you try your hand at writing them yourself.

❖ **Note** Just a reminder: All the column names and table names used in these examples are drawn from the sample database structures shown in Appendix B.

Sales Orders Database

"Show me the names of all our vendors."

Translation Select the vendor name from the vendors table

Clean Up Select ~~the~~ vendor name from ~~the~~ vendors ~~table~~

SQL

```
SELECT VendName
FROM Vendors
```

CH04_Vendor_Names (10 Rows)

VendName
Shinoman, Incorporated
Viscount
Nikoma of America
ProFormance
Kona, Incorporated
Big Sky Mountain Bikes
Dog Ear
Sun Sports Suppliers
Lone Star Bike Supply
Armadillo Brand

"What are the names and prices of all the products we carry?"

Translation Select product name, retail price from the products table

Clean Up Select product name, retail price from ~~the~~ products ~~table~~

SQL

```
SELECT ProductName, RetailPrice
FROM Products
```

CH04_Product_Price_List (40 Rows)

ProductName	Retail Price
Trek 9000 Mountain Bike	$1,200.00
Eagle FS-3 Mountain Bike	$1,800.00
Dog Ear Cyclecomputer	$75.00
Victoria Pro All Weather Tires	$54.95
Dog Ear Helmet Mount Mirrors	$7.45
Viscount Mountain Bike	$635.00
Viscount C-500 Wireless Bike Computer	$49.00
Kryptonite Advanced 2000 U-Lock	$50.00
Nikoma Lok-Tight U-Lock	$33.00
Viscount Microshell Helmet	$36.00
<< *more rows here* >>	

"Which states do our customers come from?"

Translation Select the distinct state values from the customers table

Clean Up Select ~~the~~ distinct state ~~values~~ from ~~the~~ customers ~~table~~

SQL
```
SELECT DISTINCT CustState
FROM Customers
```

CH04_Customer_States (4 Rows)

CustState
CA
OR
TX
WA

Entertainment Agency Database

"List all entertainers and the cities they're based in, and sort the results by city and name in ascending order."

Translation Select city and stage name from the entertainers table and order by city and stage name

Clean Up Select city ~~and~~ stage name from ~~the~~ entertainers ~~table and~~ order by city ~~and~~ stage name

SQL

```
SELECT EntCity, EntStageName
FROM Entertainers
ORDER BY EntCity ASC, EntStageName ASC
```

CH04_Entertainer_Locations (13 Rows)

EntCity	EntStageName
Auburn	Caroline Coie Cuartet
Auburn	Topazz
Bellevue	Jazz Persuasion
Bellevue	Jim Glynn
Bellevue	Susan McLain
Redmond	Carol Peacock Trio
Redmond	JV & the Deep Six
Seattle	Coldwater Cattle Company
Seattle	Country Feeling
Seattle	Julia Schnebly
<< more rows here >>	

"Give me a unique list of engagement dates. I'm not concerned with how many engagements there are per date."

Translation Select the distinct start date values from the engagements table

Clean Up Select ~~the~~ distinct start date ~~values~~ from ~~the~~ engagements ~~table~~

SQL
```
SELECT DISTINCT StartDate
FROM Engagements
```

CH04_Engagement_Dates (64 Rows)

StartDate
2007-09-01
2007-09-10
2007-09-11
2007-09-15
2007-09-17
2007-09-18
2007-09-24
2007-09-29
2007-09-30
2007-10-01
<< *more rows here* >>

School Scheduling Database

"Can we view complete class information?"

Translation Select all columns from the classes table

Clean Up Select ~~all columns~~ * from ~~the~~ classes ~~table~~

SQL
```
SELECT *
FROM Classes
```

CH04_Class_Information (76 Rows)

ClassID	SubjectID	ClassRoomID	Credits	StartTime	Duration	<<other columns>>
1000	11	1231	5	10:00	50	...
1002	12	1619	4	15:30	110	...
1004	13	1627	4	08:00	50	...
1006	13	1627	4	09:00	110	...
1012	14	1627	4	13:00	170	...
1020	15	3404	4	13:00	110	...
1030	16	1231	5	11:00	50	...
1031	16	1231	5	14:00	50	...
1156	37	3443	5	08:00	50	...
1162	37	3443	5	09:00	80	...
<< *more rows here* >>						

"Give me a list of the buildings on campus and the number of floors for each building. Sort the list by building in ascending order."

Translation Select building name and number of floors from the buildings table, ordered by building name

Clean Up Select building name ~~and~~ number of floors from ~~the~~ buildings ~~table,~~ order~~ed~~ by building name

SQL
```
SELECT BuildingName, NumberOfFloors
FROM Buildings
ORDER BY BuildingName ASC
```

CH04_Building_List (6 Rows)

BuildingName	NumberOfFloors
Arts and Sciences	3
College Center	3
Instructional Building	3
Library	2
PE and Wellness	1
Technology Building	2

Bowling League Database

"Where are we holding our tournaments?"

Translation Select the distinct tourney location values from the tournaments table

Clean Up Select ~~the~~ distinct tourney location ~~values~~ from ~~the~~ tournaments ~~table~~

SQL

```
SELECT DISTINCT TourneyLocation
FROM Tournaments
```

CH04_Tourney_Locations (7 Rows)

TourneyLocation
Acapulco Lanes
Bolero Lanes
Imperial Lanes
Red Rooster Lanes
Sports World Lanes
Thunderbird Lanes
Totem Lanes

"Give me a list of all tournament dates and locations. I need the dates in descending order and the locations in alphabetical order."

Translation — Select tourney date and location from the tournaments table and order by tourney date in descending order and location in ascending order

Clean Up — Select tourney date ~~and~~ location from ~~the~~ tournaments ~~table and~~ order by tourney date ~~in~~ descending ~~order and~~ location ~~in~~ ascending ~~order~~

SQL

```
SELECT TourneyDate, TourneyLocation
FROM Tournaments
ORDER BY TourneyDate DESC, TourneyLocation ASC
```

CH04_Tourney_Dates (14 Rows)

TourneyDate	TourneyLocation
2008-08-15	Totem Lanes
2008-08-08	Imperial Lanes
2008-08-01	Sports World Lanes
2008-07-25	Bolero Lanes
2008-07-18	Thunderbird Lanes
2008-07-11	Red Rooster Lanes
2007-12-04	Acapulco Lanes
2007-11-27	Totem Lanes
2007-11-20	Sports World Lanes
2007-11-13	Imperial Lanes
<< more rows here >>	

Recipes Database

"What types of recipes do we have, and what are the names of the recipes we have for each type? Can you sort the information by type and recipe name?"

Translation: Select recipe class ID and recipe title from the recipes table and order by recipe class ID and recipe title

Clean Up: Select recipe class ID ~~and~~ recipe title from ~~the~~ recipes ~~table~~ ~~and~~ order by recipeclass ID ~~and~~ recipe title

SQL:

```
SELECT RecipeClassID, RecipeTitle
FROM Recipes
ORDER BY RecipeClassID ASC, RecipeTitle ASC
```

CH04_Recipe_Classes_And_Titles (15 Rows)

RecipeClassID	RecipeTitle
1	Fettuccini Alfredo
1	Huachinango Veracruzana (Red Snapper, Veracruz style)
1	Irish Stew
1	Pollo Picoso
1	Roast Beef
1	Salmon Filets in Parchment Paper
1	Tourtière (French-Canadian Pork Pie)
2	Asparagus
2	Garlic Green Beans
3	Yorkshire Pudding
<< more rows here >>	

"Show me a list of unique recipe class IDs in the recipes table."

Translation	Select the distinct recipe class ID values from the recipes table
Clean Up	Select ~~the~~ distinct recipe class ID ~~values~~ from ~~the~~ recipes ~~table~~
SQL	`SELECT DISTINCT RecipeClassID` `FROM Recipes`

CH04_Recipe_Class_Ids (6 Rows)

RecipeClassID
1
2
3
4
5
6

SUMMARY

In this chapter, we introduced the SELECT operation, and you learned that it is one of four data manipulation operations in SQL. (The others are UPDATE, INSERT, and DELETE, covered in Part V.) We also discussed how the SELECT operation can be divided into three smaller operations: the SELECT statement, the SELECT expression, and the SELECT query.

The discussion then turned to the SELECT statement, where you were introduced to its component clauses. We covered the fact that the SELECT and FROM clauses are the fundamental clauses required to retrieve information from the database and that the remaining clauses—WHERE, GROUP BY, and HAVING—are used to conditionally process and filter the information returned by the SELECT clause.

We briefly diverged into a discussion of the difference between data and information. You learned that the values stored in the database are data and that information is data that has been processed in a manner that makes it meaningful to the person viewing it. You also learned that the rows of information returned by a SELECT statement are known as a result set.

Retrieving information was the next topic of discussion, and we began by presenting the basic form of the SELECT statement. You learned how to build a proper SELECT statement by using a three-step technique that involves taking a request and translating it into proper SQL syntax. You also learned that you could use two or more columns in the SELECT clause to expand the scope of information you retrieve from your database. We followed this section with a quick look at the DISTINCT keyword, which you learned is the means for eliminating duplicate rows from a result set.

Next, we looked at the SELECT query and how it can be combined with a SELECT statement to sort the SELECT statement's result set. You learned that this is necessary because the SELECT query is the only SELECT operation that contains an ORDER BY clause. We went on to show that the ORDER BY clause is used to sort the information by one or more columns and that each column can have its own ascending or descending sort specification. A brief discussion on saving your SELECT statements followed, and you learned that you can save your statement as a query or a view for future use.

Finally, we presented a number of examples using various tables in the sample databases. The examples illustrated how the various concepts and techniques presented in this chapter are used in typical scenarios and applications. In the next chapter, we'll take a closer look at the SELECT clause and show you how to retrieve something besides information from a list of columns.

The following section presents a number of requests that you can work out on your own.

Problems for You to Solve

Below, we show you the request statement and the name of the solution query in the sample databases. If you want some practice, you can work out the SQL you need for each request and then check your answer with the query we saved in the samples. Don't worry if your syntax doesn't exactly match the syntax of the queries we saved—as long as your result set is the same.

Sales Orders Database

1. *"Show me all the information on our employees."*
   You can find the solution in CH04_Employee_Information (8 rows).

2. *"Show me a list of cities, in alphabetical order, where our vendors are located, and include the names of the vendors we work with in each city."*
   You can find the solution in CH04_Vendor_Locations (10 rows).

Entertainment Agency Database

1. *"Give me the names and phone numbers of all our agents, and list them in last name/first name order."*
   You can find the solution in CH04_Agent_Phone_List (9 rows).
2. *"Give me the information on all our engagements."*
   You can find the solution in CH04_Engagement_Information (111 rows).
3. *"List all engagements and their associated start dates. Sort the records by date in descending order and by engagement in ascending order."*
   You can find the solution in CH04_Scheduled_Engagements (111 rows).

School Scheduling Database

1. *"Show me a complete list of all the subjects we offer."*
   You can find the solution in CH04_Subject_List (56 rows).
2. *"What kinds of titles are associated with our faculty?"*
   You can find the solution in CH04_Faculty_Titles (3 rows).
3. *"List the names and phone numbers of all our staff, and sort them by last name and first name."*
   You can find the solution in CH04_Staff_Phone_List (27 rows).

Bowling League Database

1. *"List all of the teams in alphabetical order."*
   You can find the solution in CH04_Team_List (8 rows).
2. *"Show me all the bowling score information for each of our members."*
   You can find the solution in CH04_Bowling_Score_Information (1,344 rows).
3. *"Show me a list of bowlers and their addresses, and sort it in alphabetical order."*
   You can find the solution in CH04_Bowler_Names_Addresses (32 rows).

Recipes Database

1. *"Show me a list of all the ingredients we currently keep track of."*
   You can find the solution in CH04_Complete_Ingredients_List (79 rows).
2. *"Show me all the main recipe information, and sort it by the name of the recipe in alphabetical order."*
   You can find the solution in CH04_Main_Recipe_Information (15 rows).