

CHAPTER 5

Introducing SQL

IN THIS CHAPTER

Understanding Data Sources	90
Preparing to Write SQL Queries	91
Creating Queries	92
Sorting Query Results	95
Filtering Data	97

SQL, pronounced *seque1* or *S-Q-L*, is an acronym for Structured Query Language. SQL is a language you use to access and manipulate data in a relational database. It is designed to be both easy to learn and extremely powerful, and its mass acceptance by so many database vendors proves that it has succeeded in both.

In 1970, Dr. E. F. Codd, the man credited with being the father of the relational database, described a universal language for data access. In 1974, engineers at IBM's San Jose Research Center created the Structured English Query Language, or SEQUEL, built on Codd's ideas. This language was incorporated into System R, IBM's pioneering relational database system.

Toward the end of the 1980s, two of the most important standards bodies, the American National Standards Institute (ANSI) and the International Standards Organization (ISO), published SQL standards, opening the door to mass acceptance. With these standards in place, SQL was poised to become the de-facto standard used by every major database vendor.

Although SQL has evolved a great deal since its early SEQUEL days, the basic language concepts and its founding premises have remained the same. The beauty of SQL is its simplicity. But don't let that simplicity deceive you. SQL is a powerful language, and it encourages you to be creative in your problem solving. You can almost always find more than one way to perform a complex query or to extract desired data. Each solution has pros and cons, and no solution is explicitly right or wrong.

Before you panic at the thought of learning a new language, let me reassure you that SQL really is easy to learn. In fact, you need to learn only four statements to be able to perform almost all the data manipulation you will need on a regular basis. Table 5.1 lists these statements.

Table 5.1 SQL-Based Data Manipulation Statements

STATEMENT	DESCRIPTION
SELECT	Queries a table for specific data.
INSERT	Adds new data to a table.
UPDATE	Updates existing data in a table.
DELETE	Removes data from a table.

Each of these statements takes one or more keywords as parameters. By combining various statements and keywords, you can manipulate your data in as many ways as you can imagine.

ColdFusion provides you with all the tools you need to add Web-based interaction to your databases. ColdFusion itself, though, has no built-in database. Instead, it communicates with whatever database you select, passing updates and requests and returning query results.

TIP

This chapter (and the next) is by no means a complete SQL tutorial, so a good book on SQL is a must for ColdFusion developers. If you want a crash course on all the major SQL language elements, you might want to pick a copy of my “Sams Teach Yourself SQL in 10 Minutes” (ISBN: 0672321289).

Understanding Data Sources

Back in Chapter 3, “Accessing the ColdFusion Administrator,” you created a data source which you’ll now use to experiment with SQL. But before doing so, it is worth taking a moment to understand data sources, databases, and how they relate to each.

As explained in Chapter 2, “Building the Databases,” a database is a collection of tables that store related data. Databases are generally used in one of two ways:

- Directly within a DBMS application, for example, Microsoft Access or SQL Server’s Enterprise Manager. These applications tend to be very database specific (they are usually designed by the database vendor for use with specific databases).
- Via third party applications, commercial or custom, that know how to interact with existing external databases.

ColdFusion MX is the latter. ColdFusion is not a database product. It does, however, let you write applications that will interact with any and all databases.

How do third party applications interact with databases (usually created by other vendors)? That’s where data sources come in to the picture, but first we need to look at the *database driver*. Almost every database out there has available database drivers—special bits of software that provide access to the databases. Each database product requires its own driver (the Oracle driver, for example, will not work for SQL Server), although a single driver can support multiple databases (the same SQL Server driver can access many different SQL Server installations).

There are two primary standards for databases drivers:

- ODBC has been around for a long time, and is one of the most widely used database driver standards. ODBC is primarily used on Windows, although other platforms are supported too.
- JDBC is Java's database driver implementation, and is supported on all platforms and environments running Java.

NOTE

ColdFusion 5 and earlier used ODBC database drivers. ColdFusion MX, which is Java based, uses JDBC instead.

Regardless of the database driver or standard used, the purpose of the driver remains the same—database drivers conceal databases differences providing simplified access to databases. For example, the internal workings of Microsoft Access and Oracle are very different, but when accessed via a database driver they look the same (or at least more alike). This allows the same application to interact with all sorts of databases, without needing to be customized or modified for each. Database drivers are very database specific, and in being so access to databases need not be database specific at all.

Of course, different database drivers need different information—for example, the Microsoft Access driver simply needs to know the name and location of the MDB file to use, whereas the Oracle and SQL Server database drivers need to know server information and well as an account login and password.

This driver specific information could be provided each time it is needed, or a data source could be created. A data source is simply a driver plus any related information stored for future use. Client applications, like ColdFusion, use data sources to interact with databases.

Preparing to Write SQL Queries

Now that you have a data source, all you need is a client application with which to access the data. Ultimately, the client you will use is ColdFusion via CFML code; after all, that is why you're reading this book. But to start learning SQL, well use something simpler, a SQL Query Tool (written in ColdFusion). The tool is named `sql.cfm` and is in a directory named `sql` under the `ows` directory, and so the path to it (if using the integrated Web server) will be:

```
http://localhost:8500/ows/sql/sql.cfm
```

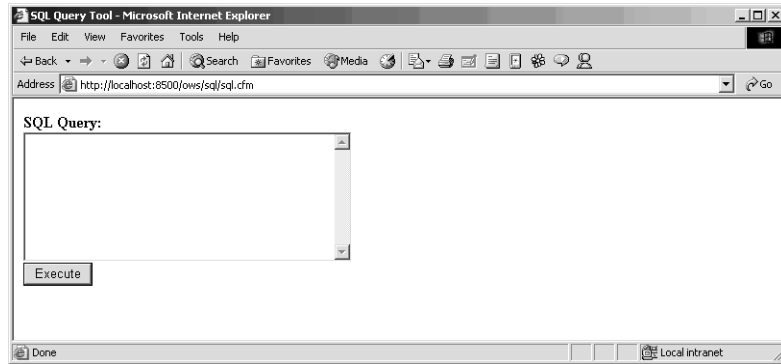
The SQL Query Tool, shown in Figure 5.1, allows you to enter SQL statements in the box provided which are executed when the Execute button is clicked. Results, if there are any, are displayed in the bottom half of the screen.

CAUTION

The SQL Query Tool is intended for use on development computers and should not be installed on live (production) servers.

Figure 5.1

The SQL Query Tool allows SQL statements to be entered manually and then executed.

**NOTE**

The SQL Query Tool allows SQL statements to be executed against databases. As a rule, this type of tool is dangerous as it could be used to delete or change data (accidentally or maliciously). To help prevent this from occurring SQL Query Tool has several built in security measures; by default it only allows **SELECT** statements, it has a hardcoded data source, and it only allows SQL statements to be executed locally (local IP address only). To use SQL Query Tool remotely you must explicitly allow your own IP address access to the tool by modifying the `Application.cfm` file specifying the address in the `ip_restrict` variable.

NOTE

Readers of prior editions of this book will note that in the past tools like MS-Query were used in these SQL chapters. As ColdFusion MX is Java based, and thus uses JDBC, data sources created using the ColdFusion Administrator will not be accessible via MS-Query or any other ODBC based tool. As such, a ColdFusion based SQL Query Tool is being used instead.

Creating Queries

With all the preliminaries taken care of, you can roll up your sleeves and start writing SQL. The SQL statement you will use most is the **SELECT** statement. You use **SELECT**, as its name implies, to select data from a table.

Most **SELECT** statements require at least the following two parameters:

- What data you want to select, known as the select list. If you specify more than one item, you must separate each with a comma.
- The table (or tables) from which to select the data, specified with the **FROM** keyword.

The first SQL **SELECT** you will create is a query for a list of movies in the `Films` table. Type the code in Listing 5.1 as seen in Figure 5.2, and then execute the statement by clicking the Execute button.

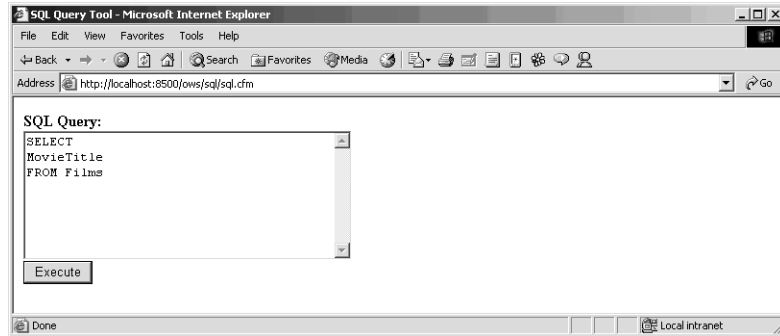
Listing 5.1 Simple **SELECT** Statement

```
SELECT
MovieTitle
FROM Films
```

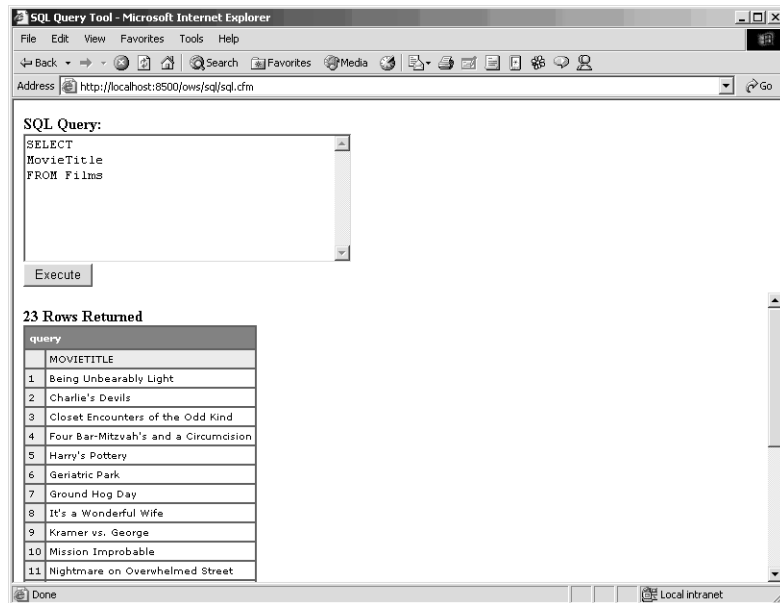
That's it! You've written your first SQL statement. The results will be shown as seen in Figure 5.3.

Figure 5.2

SQL statements are entered in the SQL Query Tool's SQL Query field.

**Figure 5.3**

The SQL Query Tool displays query results in the bottom half of the screen.

**TIP**

You can enter SQL statements on one long line or break them up over multiple lines. All whitespace characters (spaces, tabs, new-line characters) are ignored when the command is processed. If you break a statement into multiple lines and indent parameters, you make the statement easier to read and debug.

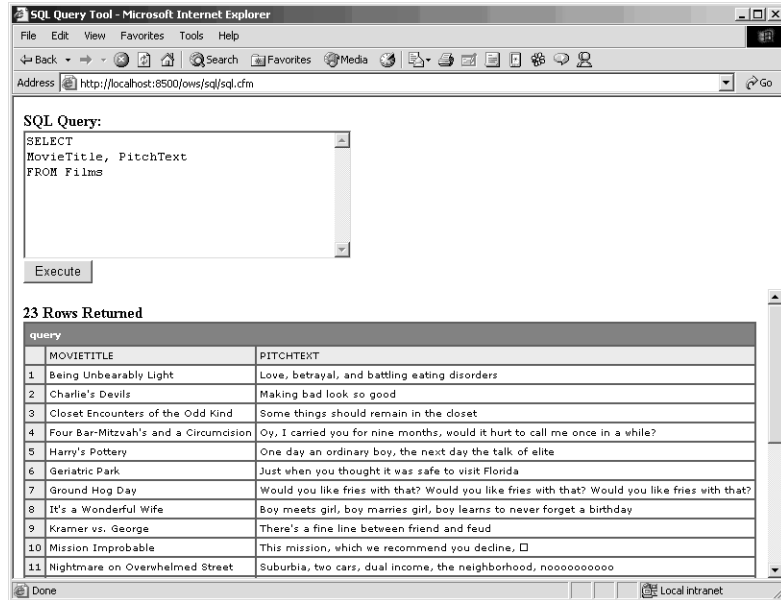
Here is another example. Type the code in Listing 5.2 and then click the Execute button to display two columns as seen in Figure 5.4.

Listing 5.2 Multi-column SELECT Statement

```
SELECT
MovieTitle, PitchText
FROM Films
```

Figure 5.4

The SQL Query Tool displays the results of all specified columns.



Before you go any further, take a closer look at the SQL code in Listing 5.2. The first parameter you pass to the `SELECT` statement is a list of the two columns you want to see. A column is specified by its name (for example, `MovieTitle`) or as `table.column` (such as `Films.MovieTitle`, where `Films` is the table name and `MovieTitle` is the column name).

Because you want to specify two columns, you must separate them with commas. No comma appears after the last column name, so if you have only one column in your select list, you don't need a comma.

Right after the select list, you specify the table on which you want to perform the query. You always precede the table name with the keyword `FROM`. The table is specified by name, in this case `Films`.

NOTE

SQL statements are not case sensitive; that is, you can specify the `SELECT` statement as `SELECT`, `select`, `Select`, or however you want. Common practice, however, is to enter all SQL keywords in uppercase and parameters in lowercase or mixed case. This way, you can read the SQL code and spot typos more easily.

Now modify the `SELECT` statement so it looks like the code in Listing 5.3; then execute it.

Listing 5.3 SELECT All Columns

```
SELECT
*
FROM Films
```

This time, instead of specifying explicit columns to select, you use an asterisk (*). The asterisk is a special select list option that represents all columns. The data pane now shows all the columns in the table in the order in which they are returned by the database table itself.

CAUTION

Generally, you should not use an asterisk in the select list unless you really need every column. Each column you select requires its own processing, and retrieving unnecessary columns can dramatically affect retrieval times as your tables get larger.

Sorting Query Results

When you use the SELECT statement, the results are returned to you in the order in which they appear in the table. This is usually the order in which the rows were added to the table, typically not a sort order that is of much use to you. More often than not, when you retrieve data using a SELECT statement, you want to sort the query results. To sort rows, you need to add the ORDER BY clause. ORDER BY always comes after the table name; if you try to use it before, you generate a SQL error.

Now click the SQL button, enter the SQL code shown in Listing 5.4, and then click OK.

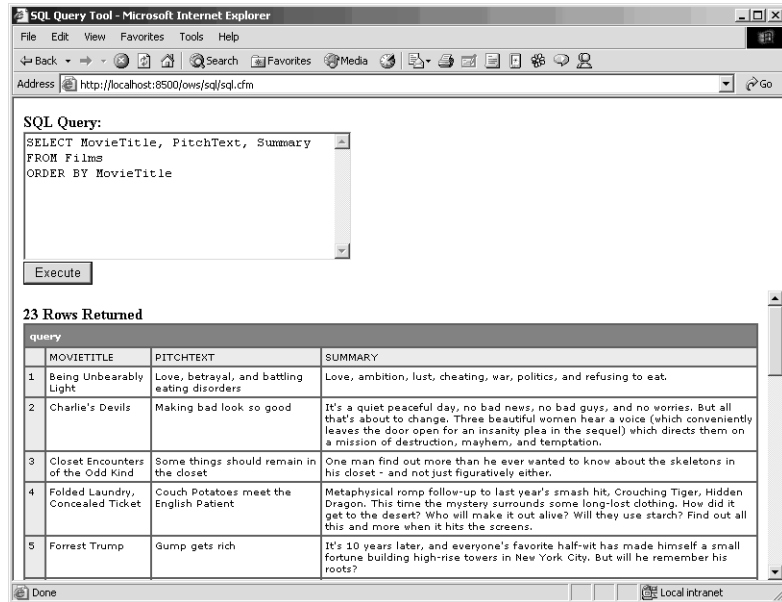
Listing 5.4 SELECT with Sorted Output

```
SELECT MovieTitle, PitchText, Summary
FROM Films
ORDER BY MovieTitle
```

Your output is then sorted by the MovieTitle column, as shown in Figure 5.5.

Figure 5.5

You use the ORDER BY clause to sort SELECT output.



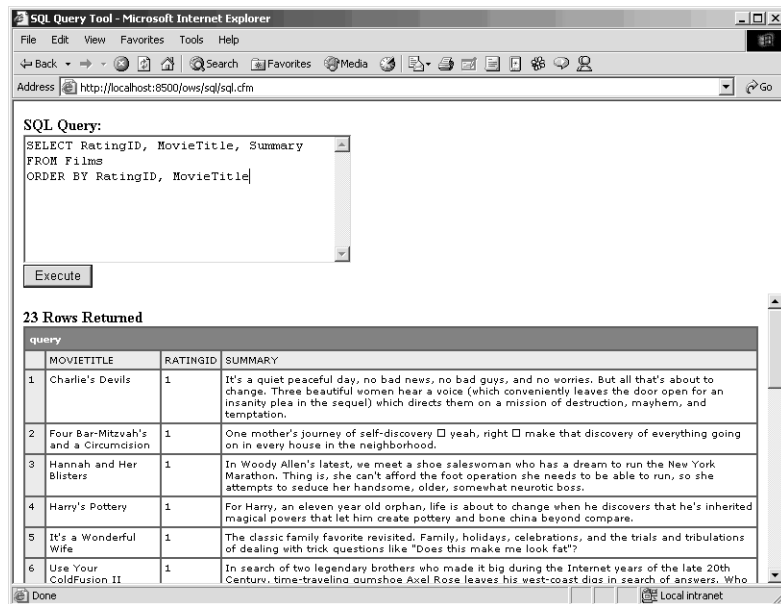
What if you need to sort by more than one column? No problem. You can pass multiple columns to the `ORDER BY` clause. Once again, if you have multiple columns listed, you must separate them with commas. The SQL code in Listing 5.5 demonstrates how to sort on more than one column by sorting by `RatingID`, and then by `MovieTitle` within each `RatingID`. The sorted output is shown in Figure 5.6.

Listing 5.5 SELECT with Output Sorted on More Than One Column

```
SELECT RatingID, MovieTitle, Summary
FROM Films
ORDER BY RatingID, MovieTitle
```

Figure 5.6

You can sort output by more than one column via the `ORDER BY` clause.



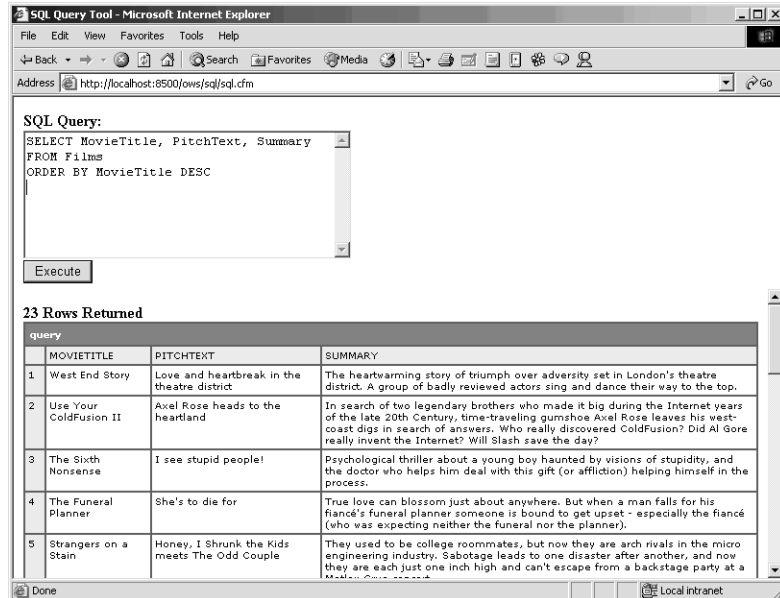
You also can use `ORDER BY` to sort data in descending order (Z–A). To sort a column in descending order, just use the `DESC` (short for descending) parameter. Listing 5.6 retrieves all the movies and sorts them by title in reverse order. Figure 5.7 shows the output that this SQL `SELECT` statement generates.

Listing 5.6 SELECT with Output Sorted in Reverse Order

```
SELECT MovieTitle, PitchText, Summary
FROM Films
ORDER BY MovieTitle DESC
```

Figure 5.7

Using the `ORDER BY` clause, you can sort data in a descending sort sequence.



Filtering Data

So far, all your queries have retrieved all the rows in the table. You also can use the `SELECT` statement to retrieve only data that matches specific search criteria. To do so, you must use the `WHERE` clause and provide a restricting condition. If a `WHERE` clause is present, when the SQL `SELECT` statement is processed, every row is evaluated against the condition. Only rows that pass the restriction are selected.

If you use a `WHERE` clause, it must appear after the table name. If you use both the `ORDER BY` and `WHERE` clauses, the `WHERE` clause must appear after the table name but before the `ORDER BY` clause.

Filtering on a Single Column

To demonstrate filtering, modify the `SELECT` statement to retrieve only movies with a `RatingID` of 1. Listing 5.7 contains the `SELECT` statement, and the resulting output is shown in Figure 5.8.

Listing 5.7 `SELECT` with `WHERE` Clause

```
SELECT MovieTitle, PitchText, Summary
FROM Films
WHERE RatingID=1
ORDER BY MovieTitle DESC
```

Figure 5.8

Using the WHERE clause, you can restrict the scope of a SELECT search.

The screenshot shows the SQL Query Tool interface. The query entered is:

```
SELECT MovieTitle, PitchText, Summary
FROM Films
WHERE RatingID=1
ORDER BY MovieTitle DESC
```

The results table, titled "7 Rows Returned", contains the following data:

query	MOVIE TITLE	PITCHTEXT	SUMMARY
1	West End Story	Love and heartbreak in the theatre district	The heartwarming story of triumph over adversity set in London's theatre district. A group of badly reviewed actors sing and dance their way to the top.
2	Use Your ColdFusion II	Axel Rose heads to the heartland	In search of two legendary brothers who made it big during the Internet years of the late 20th Century, time-traveling gumshoe Axel Rose leaves his west-coast digs in search of answers. Who really discovered ColdFusion? Did Al Gore really invent the Internet? Will Slash save the day?
3	It's a Wonderful Wife	Boy meets girl, boy marries girl, boy learns to never forget a birthday	The classic family favorite revisited. Family, holidays, celebrations, and the trials and tribulations of dealing with trick questions like "Does this make me look fat?"
4	Harry's Pottery	One day an ordinary boy, the next day the talk of elite	For Harry, an eleven year old orphan, life is about to change when he discovers that he's inherited magical powers that let him create pottery and bone china beyond compare.
5	Hannah and Her Blisters	Annie Hall meets Chariots of Fire	In Woody Allen's latest, we meet a shoe salesman who has a dream to run the New York Marathon. Thing is, she can't afford the foot operation she needs to be able to run, so she attempts to seduce her handsome, older, somewhat neurotic...

Filtering on Multiple Columns

The WHERE clause also can take multiple conditions. To search for Ben Forta, you can specify a search condition in which the first name is Ben and the last name is Forta, as shown in Listing 5.8. As Figure 5.9 shows, only Ben Forta is retrieved.

Listing 5.8 SELECT with Multiple WHERE Clauses

```
SELECT FirstName, LastName, Email
FROM Contacts
WHERE FirstName='Ben' AND LastName='Forta'
```

Figure 5.9

You can narrow your search with multiple WHERE clauses.

The screenshot shows the SQL Query Tool interface. The query entered is:

```
SELECT FirstName, LastName, Email
FROM Contacts
WHERE FirstName='Ben'
AND LastName='Forta'
```

The results table, titled "1 Row Returned", contains the following data:

query	EMAIL	FIRSTNAME	LASTNAME
1	ben@forta.com	Ben	Forta

CAUTION

Text passed to a SQL query must be enclosed within quotation marks. If you omit the quotation marks, the SQL parser thinks that the text you specified is the name of a column, and you receive an error because that column does not exist. Pure SQL allows strings to be enclosed within single quotation marks ("like this") or within double quotation marks ("like this"). But when passing text in a SQL statement to an ODBC or JDBC driver, you *must* use single quotation marks. If you use double quotation marks, the parser treats the first double quotation mark as a statement terminator, ignoring all text after it.

The AND and OR Operators

Multiple WHERE clauses can be evaluated as AND conditions or OR conditions. The example in Listing 5.8 is an AND condition. Only rows in which both the last name is Forta and the first name is Ben will be retrieved. If you change the clause to the following, contacts with a first name of Ben will be retrieved (regardless of last name) and contacts with a last name of Forta will be retrieved (regardless of first name):

```
WHERE FirstName='Ben' OR LastName='Forta'
```

You can combine the AND and OR operators to create any search condition you need. Listing 5.9 shows a WHERE clauses that can be used to retrieve only Ben Forta and Rick Richards.

Listing 5.9 Combining WHERE Clauses with AND and OR Operators

```
SELECT NameFirst, NameLast, Email
FROM Contacts
WHERE FirstName='Ben' AND LastName='Forta'
   OR FirstName='Rick' AND LastName='Richards'
```

Evaluation Precedence

When a WHERE clause is processed, the operators are evaluated in the following order of precedence:

- Parentheses have the highest precedence.
- The AND operator has the next level of precedence.
- The OR operator has the lowest level of precedence.

What does this mean? Well, look at the WHERE clause in Listing 5.9. The clause reads WHERE FirstName='Ben' AND LastName='Forta' OR FirstName='Rick' AND LastName='Richards'. AND is evaluated before OR so this statement looks for Ben Forta and Rick Richards, which is what we wanted.

But what would a WHERE clause of WHERE FirstName='Rick' OR FirstName='Ben' AND LastName='Forta' return? Does that statement mean *anyone whose first name is either Rick or Ben, and whose last name is Forta*, or does it mean *anyone whose first name is Rick, and also Ben Forta*? The difference is subtle, but if the former is true then only contacts with a last name of Forta will be retrieved, whereas if the latter is true then any Rick will be retrieved, regardless of last name.

So which is it? Because AND is evaluated first, the clause means *anyone whose first name is Rick, and also Ben Forta*, which might be what you want. And then again, it might not.

To prevent the ambiguity created by mixing AND and OR statements, parentheses are used to group related statements. Parentheses have a higher order of evaluation than both AND and OR, so they can be used to explicitly match related clauses. Consider the following WHERE clauses:

```
WHERE (FirstName='Rick' OR FirstName='Ben') AND (LastName='Forta')
```

This clause means *anyone whose first name is either Rick or Ben, and whose last name is Forta*.

```
WHERE (FirstName='Rick') OR (FirstName='Ben' AND LastName='Forta')
```

This clause means *anyone whose first name is Rick, and also Ben Forta*.

As you can see, the exact same set of WHERE clauses can mean very different things depending on where parentheses are used.

TIP

Always using parentheses whenever you have more than one WHERE clause is good practice. They make the SQL statement easier to read and easier to debug.

WHERE Conditions

For the examples to this point, you have used only the = (equal) operator. You filtered rows based on their being equal to a specific value. Many other operators and conditions can be used with the WHERE clause; they're listed in Table 5.2.

Table 5.2 WHERE Clause Search Conditions

CONDITION	DESCRIPTION
=	Equal to. Tests for equality.
<>	Not equal to. Tests for inequality.
<	Less than. Tests that the value on the left is less than the value on the right.
<=	Less than or equal to. Tests that the value on the left is less than or equal to the value on the right.
>	Greater than. Tests that the value on the left is greater than the value on the right.
>=	Greater than or equal to. Tests that the value on the left is greater than or equal to the value on the right.
BETWEEN	Tests that a value is in the range between two values; the range is inclusive.
EXISTS	Tests for the existence of rows returned by a subquery.
IN	Tests to see whether a value is contained within a list of values.
IS NULL	Tests to see whether a column contains a NULL value.
IS NOT NULL	Tests to see whether a column contains a non-NULL value.
LIKE	Tests to see whether a value matches a specified pattern.
NOT	Negates any test.

Feel free to experiment with different `SELECT` statements using any of the `WHERE` clauses listed here. The SQL Query tool is safe, it will not update or modify data (by default) and so there is no harm in using it to experiment to with statements and clauses.

Testing for Equality: =

You use the `=` operator to test for value inequality. The following example retrieves only contacts whose last name is Smith:

```
WHERE LastName = 'Smith'
```

Testing for Inequality: <>

You use the `<>` operator to test for value inequality. The following example retrieves only contacts whose first name is not Kim:

```
WHERE FirstName <> 'Kim'
```

Testing for Less Than: <

By using the `<` operator, you can test that the value on the left is less than the value on the right. The following example retrieves only contacts whose last name is less than C, meaning that their last name begins with an A or a B:

```
WHERE LastName < 'C'
```

Testing for Less Than or Equal To: <=

By using the `<=` operator, you can test that the value on the left is less than or equal to the value on the right. The following example retrieves actors aged 21 or less:

```
WHERE Age <= 21
```

Testing for Greater Than: >

You use the `>` operator to test that the value on the left is greater than the value on the right. The following example retrieves only movies with a rating of 3 or higher (greater than 2):

```
WHERE RatingID > 2
```

Testing for Greater Than or Equal To: >=

You use the `>=` operator to test that the value on the right is greater than or equal to the value on the left. The following example retrieves only contacts whose first name begins with the letter J or higher:

```
WHERE FirstName >= 'J'
```

BETWEEN

Using the `BETWEEN` condition, you can test whether a value falls into the range between two other values. The following example retrieves only actors aged 20 to 30. Because the test is inclusive, ages 20 and 30 are also retrieved:

```
WHERE Age BETWEEN 20 AND 30
```

The `BETWEEN` condition is actually nothing more than a convenient way of combining the `>=` and `<=` conditions. You also could specify the preceding example as follows:

```
WHERE Age >= 20 AND Age <= 30
```

The advantage of using the `BETWEEN` condition is that it makes the statement easier to read.

EXISTS

Using the `EXISTS` condition, you can check whether a subquery returns any rows.

→ Subqueries are explained in Chapter 29, "More About SQL and Queries."

IN

You can use the `IN` condition to test whether a value is part of a specific set. The set of values must be surrounded by parentheses and separated by commas. The following example retrieves contacts whose last name is Black, Jones, or Smith:

```
WHERE LastName IN ('Black', 'Jones', 'Smith')
```

The preceding example is actually the same as the following:

```
WHERE LastName = 'Black' OR LastName = 'Jones' OR LastName = 'Smith'
```

Using the `IN` condition does provide two advantages. First, it makes the statement easier to read. Second, and more importantly, you can use the `IN` condition to test whether a value is within the results of another `SELECT` statement.

IS NULL and IS NOT NULL

A `NULL` value is the value of a column that is empty. The `IS NULL` condition tests for rows that have a `NULL` value; that is, the rows have no value at all in the specified column. `IS NOT NULL` tests for rows that have a value in a specified column.

The following example retrieves all contacts whose `Email` column is left empty:

```
WHERE Email IS NULL
```

To retrieve only the contacts who do have an email address, use the following example:

```
WHERE Email IS NOT NULL
```

LIKE

Using the `LIKE` condition, you can test for string pattern matches using wildcards. Two wildcard types are supported. The `%` character means that anything from that position on is considered a match. You also can use `[]` to create a wildcard for a specific character.

The following example retrieves actors whose last name begins with the letter s. To match the pattern, a last name must have an s as the first character, and anything at all after it:

```
WHERE LastName LIKE 'S%'
```

To retrieve actors with an s anywhere in their last names, you can use the following:

```
WHERE LastName LIKE '%S%'
```

You also can retrieve just actors whose last name ends with s, as follows:

```
WHERE LastName LIKE '%S'
```

The LIKE condition can be negated with the NOT operator. The following example retrieves only actors whose last name does not begin with s:

```
WHERE LastName NOT LIKE 'S%'
```

Using the LIKE condition, you also can specify a wildcard on a single character. If you want to find all actors named Smith but are not sure whether the one you want spells his or her name Smyth, you can use the following:

```
WHERE LastName LIKE 'Sm[iy]th'
```

This example retrieves only names that start with sm, then have an i or a y, and then a final th. With this example, as long as the first two characters are sm and the last two are th, and as long as the middle character is i or y, the name is considered a match.

TIP

Using the powerful LIKE condition, you can retrieve data in many ways. But everything comes with a price, and the price here is performance. Generally, LIKE conditions take far longer to process than other search conditions, especially if you use wildcards at the beginning of the pattern. As a rule, use LIKE and wildcards only when absolutely necessary.

For even more powerful searching, LIKE may be combined with other clauses using AND and OR. And you may even include multiple LIKE clauses in a single WHERE clause.

