The Foundation Statements of T-SQL

WHAT YOU WILL LEARN IN THIS CHAPTER:

➤ How to retrieve data from your database
➤ How to insert data into tables
➤ How to update data in place

At last! You’ve finally ingested the most boring stuff. It doesn’t get any worse than basic objects and tools, does it? As it goes, you have to lay down a foundation before you can build a house. The nice thing is that the foundation is now down. Having used the clichéd example of building a house, I’m going to turn it all upside down by talking about the things that let you enjoy living in the house before you’ve even talked about the plumbing. You see, when working with databases, you have to get to know how data is going to be accessed before you can learn all that much about the best ways to store it.

This chapter covers the most fundamental *Transact-SQL (T-SQL)* statements. T-SQL is SQL Server’s own dialect of Structured Query Language (SQL). The T-SQL statements that you learn about in this chapter are:

➤ SELECT
➤ INSERT
➤ UPDATE
➤ DELETE
These four statements are the bread and butter of T-SQL. You’ll learn plenty of other statements as you go along, but these statements make up the basis of T-SQL’s Data Manipulation Language (DML). Because you’ll generally issue far more commands meant to manipulate (that is, read and modify) data than other types of commands (such as those to grant user rights or create a table), you’ll find that these will become like old friends in no time at all.

In addition, SQL provides many operators and keywords that help refine your queries. You’ll learn some of the most common of these in this chapter.

**NOTE** While T-SQL is unique to SQL Server, the statements you use most of the time are not. T-SQL is largely ANSI/ISO compliant. (The standard was originally governed by ANSI, and was later taken over by the ISO. It was ANSI long enough that people generally still refer to it as ANSI compliance.) This means that, by and large, T-SQL complies with a very wide open standard. What this means to you as a developer is that much of the SQL you’re going to learn in this book is directly transferable to other SQL-based database servers, such as Sybase (which long ago used to share the same code base as SQL Server), Oracle, DB2, and MySQL. Be aware, however, that every RDBMS has different extensions and performance enhancements that it uses, above and beyond the ANSI/ISO standard. I will try to point out the ANSI versus non-ANSI ways of doing things where applicable. In some cases, you’ll have a choice to make — performance versus portability to other RDBMS systems. Most of the time, however, the ANSI way is as fast as any other option. In such a case, the choice should be clear: stay ANSI compliant.

**GETTING STARTED WITH A BASIC SELECT STATEMENT**

If you haven’t used SQL before, or don’t feel like you’ve really understood it yet, pay attention here! The `SELECT` statement and the structures used within it are the lion’s share of all the commands you will perform with SQL Server. Let’s look at the basic syntax rules for a `SELECT` statement:

```
SELECT [ALL|DISTINCT] [TOP (<expression>) [PERCENT] [WITH TIES]]
    <column list>
    [FROM <source table(s)/view(s)>]
    [WHERE <restrictive condition>]
    [GROUP BY <column name or expression using a column in the SELECT list>]
    [HAVING <restrictive condition based on the GROUP BY results>]
    [ORDER BY <column list>]
    [[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}, XMLDATA]
    [, ELEMENTS][, BINARY base 64]]
    [OPTION (<query hint>, [, ...n])]
```

Wow — that’s a lot to decipher. Let’s look at the parts.

Copyright ©2012 John Wiley & Sons, Inc.
The SELECT Statement and FROM Clause

The verb — in this case a SELECT — is the part of the overall statement that tells SQL Server what you are doing. A SELECT indicates that you are merely reading information, as opposed to modifying it. What you are selecting is identified by an expression or column list immediately following the SELECT. You'll see what I mean by this in a moment.

Next, you add in more specifics, such as where SQL Server can find this data. The FROM statement specifies the name of the table or tables from which you want to get your data. With these, you have enough to create a basic SELECT statement. Fire up the SQL Server Management Studio and take a look at a simple SELECT statement:

```sql
SELECT * FROM INFORMATION_SCHEMA.TABLES;
```

Let’s look at what you’ve asked for here. You’ve asked to SELECT information; when you’re working in SQL Server Management Studio, you can also think of this as requesting to display information. The * may seem odd, but it actually works pretty much as * does everywhere: it’s a wildcard. When you write SELECT *, you’re telling T-SQL that you want to select every column from the table.

Next, the FROM indicates that you’ve finished writing which items to output and that you’re about to indicate the source of the information — in this case, INFORMATION_SCHEMA.TABLES.

NOTE

INFORMATION_SCHEMA is a special access path that is used for displaying metadata about your system’s databases and their contents. INFORMATION_SCHEMA has several parts that can be specified after a period, such as INFORMATION_SCHEMA.SCHEMA or INFORMATION_SCHEMA.VIEWS. These special access paths to the metadata of your system are there so you don’t have to use system tables.

TRY IT OUT Using the SELECT Statement

Let’s play around with this some more. Change the current database to be the AdventureWorks database. Recall that to do this, you need only select the AdventureWorks entry from the combo box in the toolbar at the top of the Query window in the Management Studio, as shown in Figure 3-1.
Now that you have the AdventureWorks database selected, let’s start looking at some real data from your database. Try this query:

```sql
SELECT * FROM Sales.Customer;
```

After you have that in the Query window, just click Execute on the toolbar (the F5 key is a shortcut for Execute and becomes a reflex for you) and watch SQL Server give you your results. This query lists every row of data in every column of the `Sales.Customer` table in the current database (in this case, AdventureWorks). If you didn’t alter any of the settings on your system or the data in the AdventureWorks database before you ran this query, you should see the following information if you click the Messages tab:

```
(19820 row(s) affected)
```
For a `SELECT` statement, the number shown here is the number of rows that your query returned. You can also find the same information on the right side of the status bar (found below the results pane), with some other useful information, such as the login name of the user you’re logged in as, the current database as of when the last query was run (this will persist, even if you change the database in the database dropdown box, until you run your next query in this query window), and the time it took for the query to execute.

**How It Works**

Let’s look at a few specifics of your `SELECT` statement. Notice that I capitalized `SELECT` and `FROM`. This is not a requirement of SQL Server — you could run them as `SeLeCt` and `frOM` and they would work just fine. I capitalized them purely for purposes of convention and readability. You’ll find that many SQL coders use the convention of capitalizing all commands and keywords, and then use mixed case for table, column, and non-constant variable names. The standards you choose or have forced upon you may vary, but live by at least one rule: be consistent.

**NOTE** One characteristic I share with Rob, who wrote the last edition of this book, is a tendency to get up on a soapbox from time to time; I’m standing on one now, and it’s for a good reason. Nothing is more frustrating for a person who has to read your code or remember your table names than lack of consistency. When someone looks at your code or, more important, uses your column and table names, it shouldn’t take him or her long to guess most of the way you do things just by experience with the parts that he or she has already worked with. Being consistent is one of those incredibly simple rules that has been broken to at least some degree in almost every database I’ve ever worked with. Break the trend: be consistent.

The `SELECT` is telling the Query window what you are doing, and the `*` is saying what you want (remember that `*` = every column). Then comes the `FROM`.

A `FROM` clause does just what it says — that is, it defines the place from which your data should come. Immediately following the `FROM` is the names of one or more tables. In your query, all the data came from a table called `Customer`.

Now let’s try taking a little bit more specific information. Let’s say all you want is a list of all your customers by last name:

```
SELECT LastName FROM Person.Person;
```

Your results should look something like:

```
Abbas
Abel
Abercrombie
```
Note that I’ve snipped rows out of the middle for brevity. You should have 19,972 rows. Because the last name of each customer is all that you want, that’s all that you’ve selected.

**NOTE** Many SQL writers have the habit of cutting their queries short and always selecting every column by using an * in their selection criteria. This is another one of those habits to resist. While typing in an * saves you a few moments of typing the column names that you want, it also means that more data has to be retrieved than is really necessary. In addition, SQL Server must figure out just how many columns “*” amounts to and what specifically they are. You would be surprised at just how much this can drag down your application’s performance and that of your network.

In the old days, we had to completely spell out — and therefore perfectly remember — every column name that we wanted. Nowadays you’ve got IntelliSense built into SSMS, so all you have to remember are the first few characters. You see, there’s really no good argument for using * just out of laziness anymore. In short, a good rule to live by is to select what you need — no more, no less.

Let’s try another simple query. How about:

```sql
SELECT Name FROM Production.Product;
```

Again, assuming that you haven’t modified the data that came with the sample database, SQL Server should respond by returning a list of 504 products that are available in the AdventureWorks database:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjustable Race</td>
</tr>
<tr>
<td>Bearing Ball</td>
</tr>
<tr>
<td>BB Ball Bearing</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Road-750 Black, 44</td>
</tr>
<tr>
<td>Road-750 Black, 48</td>
</tr>
<tr>
<td>Road-750 Black, 52</td>
</tr>
</tbody>
</table>

The columns that you have chosen right after your `SELECT` clause are known as the `SELECT` list. In short, the `SELECT` list is made up of the columns that you have requested be output from your query.
The WHERE Clause

Well, things are starting to get boring again, aren’t they. So let’s add in the **WHERE** clause. The **WHERE** clause allows you to place conditions on what is returned to you. What you have seen thus far is unrestricted information, in the sense that every row in the table specified has been included in your results. Unrestricted queries such as these are very useful for populating things like list boxes and combo boxes and in other scenarios when you are trying to provide a *domain listing*.

**NOTE** The columns that you have chosen right after your **SELECT** clause are known as the **SELECT list**.

**NOTE** For the purposes here, don’t confuse a domain listing with that of a Windows domain. A domain listing is an exclusive list of choices. For example, if you want someone to provide you with information about a state in the United States, you might provide him or her with a list that limits the domain of choices to just the 50 states. That way, you can be sure that the option selected will be a valid one. You will see this concept of domains further when you begin learning about database design, as well as entity versus domain constraints.

Now you’ll want to try to look for more specific information. This time you don’t want a complete listing of product names; you want information on a specific product. Try this: see if you can come up with a query that returns the name, product number, and reorder point for a product with the **ProductID** 356.

Let’s break it down and build the query one piece at a time. First, you’re asking for information to be returned, so you know that you’re looking at a **SELECT** statement. The statement of what you want indicates that you would like the product name, product number, and reorder point, so you have to know the column names for these pieces of information. You’re also going to need to know from which table or tables you can retrieve these columns.

If you haven’t memorized the database schema yet, you’ll want to explore the tables that are available. Because you’ve already used the **Production.Product** table once before, you know that it’s there. The **Production.Product** table has several columns. To give you a quick listing of the column options, you can study the Object Explorer tree of the **Production.Product** table from Management Studio. To open this screen in the Management Studio, click Tables underneath the AdventureWorks database, and then expand the **Production.Product** and Columns nodes. As in Figure 3-2, you will see each of the columns along with its data type and nullability options. Again, you’ll see some other methods of finding this information a little later in the chapter.
You don’t have a column called product name, but you do have one that’s probably what you’re looking for: Name. (Original eh?) The other two columns are, save for the missing space between the two words, just as easy to identify.

Therefore, the Products table is going to be the place you get your information FROM, and the Name, ProductNumber, and ReorderPoint columns will be the specific columns from which you’ll get your information:

\[
\text{SELECT Name, ProductNumber, ReorderPoint } \\
\text{FROM Production.Product}
\]

This query, however, still won’t give you the results that you’re after; it will still return too much information. Run it and you’ll see that it still returns every record in the table rather than just the one you want.

If the table has only a few records and all you want to do is take a quick look at it, this might be fine. After all, you can look through a small list yourself, right? But that’s a pretty big if. In any significant system, very few of your tables will have small record counts. You don’t want to have to go scrolling through 10,000 records. What if you had 100,000 or 1,000,000? Even if you felt like scrolling through them all, the time before the results were back would be increased dramatically. Finally, what do you do when you’re designing this into your application and you need a quick result that gets straight to the point?

What you’re going to need (not just here, but pretty much all the time) is a conditional statement that will limit the results of your query to exactly what you want: in this case, product identifier 356. That’s where the WHERE clause comes in. The WHERE clause immediately follows the FROM clause and defines which conditions a record has to meet before it will be returned. For your query, you want the row in which the ProductID is equal to 356. You can now finish your query:

\[
\text{SELECT Name, ProductNumber, ReorderPoint } \\
\text{FROM Production.Product } \\
\text{WHERE ProductID = 356}
\]

Run this query against the AdventureWorks database, and you should come up with the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>ProductNumber</th>
<th>ReorderPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL Grip Tape</td>
<td>GT-0820</td>
<td>600</td>
</tr>
</tbody>
</table>

(1 row(s) affected)

This time you’ve received precisely what you wanted — nothing more, nothing less. In addition, this query runs much faster than the first one.

Table 3-1 shows all the operators you can use with the WHERE clause:
### TABLE 3-1: WHERE Clause Operators

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>EXAMPLE USAGE</th>
<th>EFFECT</th>
</tr>
</thead>
</table>
| =, >, <, >=, <=, !=, !> !< | `<Column Name>` = `<Other Column Name>`  
`<Column Name>` = 'Bob' | These standard comparison operators work as they do in pretty much any programming language, with a couple of notable points: 1. What constitutes “greater than,” “less than,” and “equal to” can change depending on the collation order you have selected. (For example, "ROMEY" = "romey" in places where case-insensitive sort order has been selected, but "ROMEY" < > "romey" in a case-sensitive situation.) 2. != and <> both mean “not equal.” !< and !> mean “not less than” and “not greater than,” respectively. |
| AND, OR, NOT     | `<Column1> = <Column2> AND <Column3> >= <Column 4>  
`<Column1> != "MyLiteral" OR <Column2> = "MyOtherLiteral" | Standard Boolean logic. You can use these to combine multiple conditions into one WHERE clause. NOT is evaluated first, then AND, then OR. If you need to change the evaluation order, you can use parentheses. Note that XOR is not supported. |
| BETWEEN          | `<Column1> BETWEEN 1 AND 5` | Comparison is TRUE if the first value is between the second and third values, inclusive. It is the functional equivalent of A>=B AND A<=C. Any of the specified values can be column names, variables, or literals. |
| LIKE             | `<Column1> LIKE "ROM%"` | Uses the % and _ characters for wildcarding. % indicates a value of any length can replace the % character. _ indicates any one character can replace the _ character. Enclosing characters in [ ] symbols indicates any single character within the [ ] is OK. ([a-c] means a, b, and c are OK. [ab] indicates a or b are OK.) ^ operates as a NOT operator, indicating that the next character is to be excluded. |
| IN               | `<Column1> IN (List of Numbers)  
`<Column1> IN ("A", "b", "345")` | Returns TRUE if the value to the left of the IN keyword matches any of the values in the list provided after the IN keyword. This is frequently used in subqueries, which you will look at in Chapter 7. |
CHAPTER 3  THE FOUNDATION STATEMENTS OF T-SQL

OPERATOR | EXAMPLE USAGE | EFFECT
---|---|---
ALL, ANY, SOME | <column|expression> (comparison operator) <ANY|SOME> (subquery) | These return TRUE if any or all (depending on which you choose) values in a subquery meet the comparison operator’s (for example, <, >, =, >=) condition. ALL indicates that the value must match all the values in the set. ANY and SOME are functional equivalents and will evaluate to TRUE if the expression matches any value in the set.

EXISTS | EXISTS (subquery) | Returns TRUE if at least one row is returned by the subquery. Again, you’ll look into this one further in Chapter 7.

NOTE  Note that these are not the only operators in SQL Server. These are just the ones that apply to the WHERE clause. There are a few operators that apply only as assignment operators (rather than comparison). These are inappropriate for a WHERE clause.

ORDER BY

In the queries that you’ve run thus far, most have come out in something resembling alphabetical order. Is this by accident? It will probably come as somewhat of a surprise to you, but the answer to that is yes, it’s been purely dumb luck so far. If you don’t say you want a specific sorting on the results of a query, you get the data in the order that SQL Server decides to give it to you. This will always be based on what SQL Server decided was the lowest-cost way to gather the data. It will usually be based either on the physical order of a table or on one of the indexes SQL Server used to find your data.

NOTE  Microsoft’s samples have a nasty habit of building themselves in a manner that happen to lend themselves to coming out in alphabetical order. It’s a long story as to why, but I wish they wouldn’t do that, because most data won’t work that way. The short rendition as to why has to do with the order in which Microsoft inserts its data. Because Microsoft typically inserts it in alphabetical order, the ID columns are ordered the same as alphabetical order (just by happenstance). Because most of these tables happen to be physically sorted in ID order (you’ll learn more about physical sorting of the data in Chapter 9), the data wind up appearing alphabetically sorted. If Microsoft inserted the data in a more random name order — as would likely happen in a real-life scenario — the names would tend to come out more mixed up unless you specifically asked for them to be sorted by name.
Think of an ORDER BY clause as being a sort by. It gives you the opportunity to define the order in which you want your data to come back. You can use any combination of columns in your ORDER BY clause, as long as they are columns (or derivations of columns) found in the tables within your FROM clause.

Let’s look at this query:

```sql
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product;
```

This will produce the following results:

<table>
<thead>
<tr>
<th>Name</th>
<th>ProductNumber</th>
<th>ReorderPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjustable Race</td>
<td>AR-5381</td>
<td>750</td>
</tr>
<tr>
<td>Bearing Ball</td>
<td>BA-8327</td>
<td>750</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Road-750 Black, 48</td>
<td>BK-R19B-48</td>
<td>75</td>
</tr>
<tr>
<td>Road-750 Black, 52</td>
<td>BK-R19B-52</td>
<td>75</td>
</tr>
</tbody>
</table>

(504 row(s) affected)

As it happened, the query result set was sorted in ProductID order. Why? Because SQL Server decided that the best way to look at this data was by using an index that sorts the data by ProductID. That just happened to be what created the lowest-cost (in terms of CPU and I/O) query. Were you to run this exact query when the table has grown to a much larger size, SQL Server might choose an entirely different execution plan and therefore might sort the data differently. You could force this sort order by changing your query to this:

```sql
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
ORDER BY Name;
```

Note that the WHERE clause isn’t required. It can either be there or not be there depending on what you’re trying to accomplish. Just remember that if you do have a WHERE clause, it goes before the ORDER BY clause.

Unfortunately, that previous query doesn’t really give you anything different, so you don’t see what’s actually happening. Let’s change the query to sort the data differently — by the ProductNumber:

```sql
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
ORDER BY ProductNumber;
```
Now your results are quite different. It’s the same data, but it’s been substantially rearranged:

<table>
<thead>
<tr>
<th>Name</th>
<th>ProductNumber</th>
<th>ReorderPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjustable Race</td>
<td>AR-5381</td>
<td>750</td>
</tr>
<tr>
<td>Bearing Ball</td>
<td>BA-8327</td>
<td>750</td>
</tr>
<tr>
<td>LL Bottom Bracket</td>
<td>BB-7421</td>
<td>375</td>
</tr>
<tr>
<td>ML Bottom Bracket</td>
<td>BB-8107</td>
<td>375</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classic Vest, L</td>
<td>VE-C304-L</td>
<td>3</td>
</tr>
<tr>
<td>Classic Vest, M</td>
<td>VE-C304-M</td>
<td>3</td>
</tr>
<tr>
<td>Classic Vest, S</td>
<td>VE-C304-S</td>
<td>3</td>
</tr>
<tr>
<td>Water Bottle - 30 oz.</td>
<td>WB-H098</td>
<td>3</td>
</tr>
</tbody>
</table>

(504 row(s) affected)

SQL Server still chose the least-cost method of giving you your desired results, but the particular set of tasks it actually needed to perform changed somewhat because the nature of the query changed.

You can also do your sorting using numeric fields (note that you’re querying a new table):

```
SELECT Name, SalesPersonID
FROM Sales.Store
WHERE Name BETWEEN 'g' AND 'j'
  AND SalesPersonID > 283
ORDER BY SalesPersonID, Name DESC;
```

This one results in:

<table>
<thead>
<tr>
<th>Name</th>
<th>SalesPersonID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inexpensive Parts Shop</td>
<td>286</td>
</tr>
<tr>
<td>Ideal Components</td>
<td>286</td>
</tr>
<tr>
<td>Helpful Sales and Repair Service</td>
<td>286</td>
</tr>
<tr>
<td>Helmets and Cycles</td>
<td>286</td>
</tr>
<tr>
<td>Global Sports Outlet</td>
<td>286</td>
</tr>
<tr>
<td>Gears and Parts Company</td>
<td>286</td>
</tr>
<tr>
<td>Irregulars Outlet</td>
<td>288</td>
</tr>
<tr>
<td>Hometown Riding Supplies</td>
<td>288</td>
</tr>
<tr>
<td>Good Bicycle Store</td>
<td>288</td>
</tr>
<tr>
<td>Global Bike Retailers</td>
<td>288</td>
</tr>
<tr>
<td>Instruments and Parts Company</td>
<td>289</td>
</tr>
<tr>
<td>Instant Cycle Store</td>
<td>290</td>
</tr>
<tr>
<td>Impervious Paint Company</td>
<td>290</td>
</tr>
<tr>
<td>Hiatus Bike Tours</td>
<td>290</td>
</tr>
<tr>
<td>Getaway Inn</td>
<td>290</td>
</tr>
</tbody>
</table>

(15 row(s) affected)
Getting Started with a Basic SELECT Statement

Notice several things in this query. You’ve made use of many of the things that have been covered up to this point. You’ve combined multiple WHERE clause conditions and also have an ORDER BY clause in place. In addition, you’ve added some new twists in the ORDER BY clause.

➤ You now have an ORDER BY clause that sorts based on more than one column. To do this, you simply comma-delimited the columns you wanted to sort by. In this case, you’ve sorted first by SalesPersonID and then added a sub-sort based on Name.

➤ The DESC keyword tells SQL Server that the ORDER BY should work in descending order for the Name sub-sort rather than using the default of ascending. (If you want to explicitly state that you want it to be ascending, use ASC.)

NOTE While you'll usually sort the results based on one of the columns that you are returning, it’s worth noting that the ORDER BY clause can be based on any column in any table used in the query, regardless of whether it is included in the SELECT list.

Aggregating Data Using the GROUP BY Clause

With ORDER BY, I have kind of taken things out of order compared with how the SELECT statement reads at the top of the chapter. Let’s review the overall statement structure:

```
SELECT [TOP (<expression>) [PERCENT] [WITH TIES]] <column list>
[FROM <source table(s)/view(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [{<element>}]}, XMLDATA]
[, ELEMENTS][, BINARY base 64]]
[OPTION (<query hint>, [, ...n])]
```

Why, if ORDER BY is the sixth line, did I look at it before the GROUP BY? There are two reasons:

➤ ORDER BY is used far more often than GROUP BY, so I want you to have more practice with it.

➤ I want to make sure that you understand that you can mix and match all of the clauses after the FROM clause, as long as you keep them in the order that SQL Server expects them (as defined in the syntax definition).

The GROUP BY clause is used to aggregate information. Let’s look at a simple query without a GROUP BY. Let’s say that you want to know how many parts were ordered in a given set of orders:

```
SELECT SalesOrderID, OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672);
```

Code snippet Chap03.sql
This yields a result set of:

<table>
<thead>
<tr>
<th>SalesOrderID</th>
<th>OrderQty</th>
</tr>
</thead>
<tbody>
<tr>
<td>43660</td>
<td>1</td>
</tr>
<tr>
<td>43660</td>
<td>1</td>
</tr>
<tr>
<td>43670</td>
<td>2</td>
</tr>
<tr>
<td>43670</td>
<td>2</td>
</tr>
<tr>
<td>43670</td>
<td>1</td>
</tr>
<tr>
<td>43672</td>
<td>6</td>
</tr>
<tr>
<td>43672</td>
<td>2</td>
</tr>
<tr>
<td>43672</td>
<td>1</td>
</tr>
</tbody>
</table>

(9 row(s) affected)

Even though you’ve only asked for three orders, you’re seeing each individual line of detail from the orders. You can either get out your adding machine, or you can make use of the `GROUP BY` clause with an aggregator. In this case, you can use `SUM()`:

```sql
SELECT SalesOrderID, SUM(OrderQty) FROM Sales.SalesOrderDetail WHERE SalesOrderID IN (43660, 43670, 43672) GROUP BY SalesOrderID;
```

This gets you what you were really looking for:

<table>
<thead>
<tr>
<th>SalesOrderID</th>
</tr>
</thead>
<tbody>
<tr>
<td>43660</td>
</tr>
<tr>
<td>43670</td>
</tr>
<tr>
<td>43672</td>
</tr>
</tbody>
</table>

(3 row(s) affected)

As you would expect, the `SUM` function returns totals — but totals of what? That blank column header is distinctly unhelpful. You can easily supply an alias for your result to make that easier to consume. Let’s modify the query slightly to provide a column name for the output:

```sql
SELECT SalesOrderID, SUM(OrderQty) AS TotalOrderQty FROM Sales.SalesOrderDetail WHERE SalesOrderID IN (43660, 43670, 43672) GROUP BY SalesOrderID;
```

This gets you the same basic output, but also supplies a header to the grouped column:
NOTE If you’re just trying to get some quick results, there really is no need to alias the grouped column as you’ve done here, but many of your queries are going to be written to supply information to other elements of a larger program. The code that’s utilizing your queries will need some way of referencing your grouped column; aliasing your column to some useful name can be critical in that situation. You’ll examine aliasing a bit more shortly.

If you didn’t supply the GROUP BY clause, the SUM would have been of all the values in all of the rows for the named column. In this case, however, you did supply a GROUP BY, and so the total provided by the SUM function is the total in each group.

NOTE Note that when you use a GROUP BY clause, all the columns in the SELECT list must either be aggregates (SUM, MIN/MAX, AVG, and so on) or columns included in the GROUP BY clause. Likewise, if you are using an aggregate in the SELECT list, your SELECT list must only contain aggregates, or there must be a GROUP BY clause.

You can also group based on multiple columns. To do this you just add a comma and the next column name. Let’s say, for example, that you’re looking for the number of orders each salesperson has taken for the first 10 customers. You can use both the SalesPersonID and CustomerID columns in your GROUP BY. (I’ll explain how to use the COUNT() function shortly.)

```
SELECT CustomerID, SalesPersonID, COUNT(*)
FROM Sales.SalesOrderHeader
WHERE CustomerID <= 11010
GROUP BY CustomerID, SalesPersonID
ORDER BY CustomerID, SalesPersonID;
```

This gets you counts, but the counts are pulled together based on how many orders a given salesperson took from a given customer:

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>SalesPersonID</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000</td>
<td>NULL</td>
<td>3</td>
</tr>
<tr>
<td>11001</td>
<td>NULL</td>
<td>3</td>
</tr>
<tr>
<td>11002</td>
<td>NULL</td>
<td>3</td>
</tr>
<tr>
<td>11003</td>
<td>NULL</td>
<td>3</td>
</tr>
</tbody>
</table>

(3 row(s) affected)
Aggregates

When you consider that aggregates usually get used with a GROUP BY clause, it’s probably not surprising that they are functions that work on groups of data. For example, in one of the previous queries, you got the sum of the OrderQty column. The sum is calculated and returned on the selected column for each group defined in the GROUP BY clause — in the case of your SUM, it was just SalesOrderID. A wide range of aggregates is available, but let’s play with the most common.

**NOTE** While aggregates show their power when used with a GROUP BY clause, they are not limited to grouped queries. If you include an aggregate without a GROUP BY, the aggregate will work against the entire result set (all the rows that match the WHERE clause). The catch here is that, when not working with a GROUP BY, some aggregates can only be in the SELECT list with other aggregates — that is, they can’t be paired with a column name in the SELECT list unless you have a GROUP BY. For example, unless there is a GROUP BY, AVG can be paired with SUM, but not with a specific column.

**AVG**

This one is for computing averages. Let’s try running the order quantity query that ran before, but now you’ll modify it to return the average quantity per order, rather than the total for each order:

```sql
SELECT SalesOrderID, AVG(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

Notice that the results changed substantially:

<table>
<thead>
<tr>
<th>SalesOrderID</th>
<th>Avg(OrderQty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>43660</td>
<td>1</td>
</tr>
<tr>
<td>43670</td>
<td>1</td>
</tr>
<tr>
<td>43672</td>
<td>3</td>
</tr>
</tbody>
</table>

(3 row(s) affected)
You can check the math — on order number 43672 there were three line items totaling nine altogether (9 / 3 = 3).

**MIN/MAX**

Bet you can guess these two. Yes, these grab the minimum and maximum amounts for each grouping for a selected column. Again, let's use that same query modified for the **MIN** function:

```
SELECT SalesOrderID, MIN(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

This gives the following results:

<table>
<thead>
<tr>
<th>SalesOrderID</th>
<th>OrderQty</th>
</tr>
</thead>
<tbody>
<tr>
<td>43660</td>
<td>1</td>
</tr>
<tr>
<td>43670</td>
<td>1</td>
</tr>
<tr>
<td>43672</td>
<td>1</td>
</tr>
</tbody>
</table>

(3 row(s) affected)

Modify it one more time for the **MAX** function:

```
SELECT SalesOrderID, MAX(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

And you come up with this:

<table>
<thead>
<tr>
<th>SalesOrderID</th>
<th>OrderQty</th>
</tr>
</thead>
<tbody>
<tr>
<td>43660</td>
<td>1</td>
</tr>
<tr>
<td>43670</td>
<td>2</td>
</tr>
<tr>
<td>43672</td>
<td>6</td>
</tr>
</tbody>
</table>

(3 row(s) affected)

What if, however, you wanted both the **MIN** and the **MAX**? Simple! Just use both in your query:

```
SELECT SalesOrderID, MIN(OrderQty), MAX(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```
Now, this will yield an additional column and a bit of a problem:

<table>
<thead>
<tr>
<th>SalesOrderID</th>
<th>MinOrderQty</th>
<th>MaxOrderQty</th>
</tr>
</thead>
<tbody>
<tr>
<td>43660</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>43670</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>43672</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

(3 row(s) affected)

Can you spot the issue here? You’ve gotten back everything that you asked for, but now that you have more than one aggregate column, you have a problem identifying which column is which. Sure, in this particular example you can be sure that the columns with the largest numbers are the columns generated by the \texttt{MAX} and the smallest are generated by the \texttt{MIN}. The answer to which column is which is not always so apparent, so let’s make use of an \textit{alias}. An alias allows you to change the name of a column in the result set, and you can create it by using the \texttt{AS} keyword:

\begin{verbatim}
SELECT SalesOrderID, MIN(OrderQty) AS MinOrderQty, MAX(OrderQty) AS MaxOrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
\end{verbatim}

Now your results are somewhat easier to make sense of:

<table>
<thead>
<tr>
<th>SalesOrderID</th>
<th>MinOrderQty</th>
<th>MaxOrderQty</th>
</tr>
</thead>
<tbody>
<tr>
<td>43660</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>43670</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>43672</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

(3 row(s) affected)

It’s worth noting that the \texttt{AS} keyword is actually optional. Indeed, there was a time (prior to version 6.5 of SQL Server) when it wasn’t even a valid keyword. If you like, you can execute the same query as before, but remove the two \texttt{AS} keywords from the query — you’ll see that you wind up with exactly the same results. It’s also worth noting that you can alias any column (and even, as you’ll see in the next chapter, table names), not just aggregates.

Let’s re-run this last query, but this time skip using the \texttt{AS} keyword in some places, and alias every column:

\begin{verbatim}
SELECT SalesOrderID AS 'Order Number',
       MIN(OrderQty) MinOrderQty, MAX(OrderQty) MaxOrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
\end{verbatim}

Code snippet Chap03.sql

Available for download on Wrox.com

Copyright ©2012 John Wiley & Sons, Inc.
Despite the `AS` keyword being missing in some places, you’ve still changed the name output for every column:

<table>
<thead>
<tr>
<th>Order Number</th>
<th>MinOrderQty</th>
<th>MaxOrderQty</th>
</tr>
</thead>
<tbody>
<tr>
<td>43660</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>43670</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>43672</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

(3 row(s) affected)

**NOTE** I must admit that I usually don’t include the `AS` keyword in my aliasing, but I would also admit that it’s a bad habit on my part. I’ve been working with SQL Server since before the `AS` keyword was available and have, unfortunately, become set in my ways about it (I simply forget to use it). I would, however, strongly encourage you to go ahead and make use of this extra word. Why? Well, first, because it reads somewhat more clearly and, second, because it’s the ANSI/ISO standard way of doing things.

So then, why did I even tell you about it? Well, I got you started doing it the right way — with the `AS` keyword — but I want you to be aware of alternate ways of doing things, so you aren’t confused when you see something that looks a little different.

**COUNT(Expression|*)**

The `COUNT(*)` function is about counting the rows in a query. To begin with, let’s go with one of the most common varieties of queries:

```
SELECT COUNT(*)
FROM HumanResources.Employee
WHERE HumanResources.Employee.BusinessEntityID = 5;
```

The record set you get back looks a little different from what you’re used to in earlier queries:

```
-----------
1
(1 row(s) affected)
```

Let’s look at the differences. First, as with all columns that are returned as a result of a function call, there is no default column name. If you want there to be a column name, you need to supply an alias. Next, you’ll notice that you haven’t really returned much of anything. So what does this record set represent? It is the number of rows that matched the `WHERE` condition in the query for the table(s) in the `FROM` clause.
Just for fun, try running the query without the **WHERE** clause:

```
SELECT COUNT(*)
FROM HumanResources.Employee;
```

If you haven’t done any deletions or insertions into the **Employee** table, you should get a record set that looks something like this:

```
--------
290
(1 row(s) affected)
```

What is that number? It’s the total number of rows in the **Employee** table. This is another one to keep in mind for future use.

Now, you’re just getting started! If you look back at the header for this section (the **COUNT** section), you’ll see that there are two ways of using **COUNT**. I’ve already discussed using **COUNT** with the * option. Now it’s time to look at it with an expression — usually a column name.

First, try running the **COUNT** the old way, but against a new table:

```
SELECT COUNT(*)
FROM Person.Person;
```

This is a slightly larger table, so you get a higher **COUNT**:

```
--------
19972
(1 row(s) affected)
```

Now alter your query to **select** the count for a specific column:

```
SELECT COUNT(AdditionalContactInfo)
FROM Person.Person;
```

**NOTE** Keep this query in mind. This is a basic query that you can use to verify that the exact number of rows that you expect to be in a table and match your **WHERE** condition are indeed in there.
You’ll get a result that is a bit different from the one before:

```
-----
10
Warning: Null value is eliminated by an aggregate or other SET operation.
```

(1 row(s) affected)

This new result brings with it a question. Why, since the AdditionalContactInfo column exists for every row, is there a different COUNT for AdditionalContactInfo than there is for the row count in general? The answer is fairly obvious when you stop to think about it — there isn’t a value, as such, for the AdditionalContactInfo column in every row. In short, the COUNT, when used in any form other than COUNT(*), ignores NULL values. Let’s verify that NULL values are the cause of the discrepancy:

```
SELECT COUNT(*)
FROM Person.Person
WHERE AdditionalContactInfo IS NULL;
```

This should yield the following record set:

```
-----------
19962
```

(1 row(s) affected)

Now let’s do the math:

```
10 + 19,962 = 19,972
```

That’s 10 records with a defined value in the AdditionalContactInfo field and 19,962 rows where the value in the AdditionalContactInfo field is NULL, making a total of 19,972 rows.

**NOTE** Actually, all aggregate functions ignore NULLs except for COUNT(*). Think about this for a minute — it can have a very significant impact on your results. Many users expect NULL values in numeric fields to be treated as zero when performing averages, but a NULL does not equal zero, and as such shouldn’t be used as one. If you perform an AVG or other aggregate function on a column with NULLs, the NULL values will not be part of the aggregation unless you manipulate them into a non-NULL value inside the function (using COALESCE() or ISNULL(), for example). You’ll explore this further in Chapter 7, but beware of this when coding in T-SQL and when designing your database.

Why does it matter in your database design? Well, it can have a bearing on whether or not you decide to allow NULL values in a field because of the way that queries are likely to be run against the database and how you want your aggregates to work.
Before leaving the `COUNT` function, you had better see it in action with the `GROUP BY` clause.

**NOTE** For this next example, you’ll need to load and execute the `BuildAndPopulateEmployee2.sql` file included with the downloadable source code (you can get that from the wrox.com website).

All references to employees in the following examples should be aimed at the new `Employees2` table rather than `Employees`.

Let’s say your boss has asked you to find out the number of employees who report to each manager. The statements that you’ve done thus far would count up either all the rows in the table (`COUNT(*)`) or all the rows in the table that didn’t have null values (`COUNT(ColumnName)`). When you add a `GROUP BY` clause, these aggregators perform exactly as they did before, except that they return a count for each grouping rather than the full table. You can use this to get your number of reports:

```sql
SELECT ManagerID, COUNT(*)
FROM HumanResources.Employee2
GROUP BY ManagerID;
```

Notice that you are grouping only by the `ManagerID` — the `COUNT()` function is an aggregator and, therefore, does not have to be included in the `GROUP BY` clause.

ManagerID
----------
NULL     1
1        3
4        3
5        4

(4 row(s) affected)

Your results tell us that the manager with `ManagerID 1` has three people reporting to him or her, and that three people report to the manager with `ManagerID 4`, as well as four people reporting to `ManagerID 5`. You are also able to tell that one `Employee` record had a `NULL` value in the `ManagerID` field. This employee apparently doesn’t report to anyone (hmmm, president of the company I suspect?).

It’s probably worth noting that you, technically speaking, could use a `GROUP BY` clause without any kind of aggregator, but this wouldn’t make sense. Why not? Well, SQL Server is going to wind up doing work on all the rows in order to group them, but functionally speaking you would get the same result with a `DISTINCT` option (which you’ll look at shortly), and it would operate much faster.

Now that you’ve seen how to operate with groups, let’s move on to one of the concepts that a lot of people have problems with. Of course, after reading the next section, you’ll think it’s a snap.
Placing Conditions on Groups with the HAVING Clause

Up to now, all of the conditions have been against specific rows. If a given column in a row doesn’t have a specific value or isn’t within a range of values, the entire row is left out. All of this happens before the groupings are really even thought about.

What if you want to place conditions on what the groups themselves look like? In other words, what if you want every row to be added to a group, but then you want to say that only after the groups are fully accumulated are you ready to apply the condition. Well, that’s where the **HAVING** clause comes in.

The **HAVING** clause is used only when there is also a **GROUP BY** in your query. Whereas the **WHERE** clause is applied to each row before it even has a chance to become part of a group, the **HAVING** clause is applied to the aggregated value for that group.

Let’s start off with a slight modification to the **GROUP BY** query you used at the end of the previous section — the one that tells you the number of employees assigned to each manager’s **EmployeeID**:

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
GROUP BY ManagerID;
```

In the next chapter, you’ll learn how to put names on the **EmployeeID**s that are in the **Manager** column. For now though, just note that there appear to be three different managers in the company. Apparently, everyone reports to these three people, except for one person who doesn’t have a manager assigned — that is probably the company president (you could write a query to verify that, but instead just trust in the assumption for now).

This query doesn’t have a **WHERE** clause, so the **GROUP BY** was operating on every row in the table and every row is included in a grouping. To test what would happen to your **COUNTS**, try adding a **WHERE** clause:

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
WHERE EmployeeID != 5
GROUP BY ManagerID;
```

This yields one slight change that may be somewhat different than expected:

<table>
<thead>
<tr>
<th>Manager</th>
<th>Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

(4 row(s) affected)
No rows were eliminated from the result set, but the result for ManagerID 4 was decreased by one (what the heck does this have to do with ManagerID 5?). You see, the WHERE clause eliminated the one row where the EmployeeID was 5. As it happens, EmployeeID 5 reports to ManagerID 4, so the total for ManagerID 4 is one less (EmployeeID 5 is no longer counted for this query). ManagerID 5 is not affected, as you eliminated him or her as a report (as an EmployeeID) rather than as a manager. The key thing here is to realize that EmployeeID 5 is eliminated before the GROUP BY was applied.

I want to look at things a bit differently though. See if you can work out how to answer the following question: Which managers have more than three people reporting to them? You can look at the query without the WHERE clause and tell by the COUNT, but how do you tell programmatically? That is, what if you need this query to return only the managers with more than three people reporting to them? If you try to work this out with a WHERE clause, you'll find that there isn't a way to return rows based on the aggregation. The WHERE clause is already completed by the system before the aggregation is executed. That's where the HAVING clause comes in:

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
WHERE EmployeeID != 5
GROUP BY ManagerID
HAVING COUNT(*) > 3;
```

Try it out and you'll come up with something a little bit more like what you were after:

```
Manager     Reports  
-----------  -------  
5           4       
(1 row(s) affected)
```

There is only one manager who has more than three employees reporting to him or her.

### Outputting XML Using the FOR XML Clause

SQL Server has a number of features to natively support XML. From being able to index XML data effectively to validating XML against a schema document, SQL Server is very robust in meeting XML data storage and manipulation needs.

One of the oldest features in SQL Server’s XML support arsenal is the FOR XML clause you can use with the SELECT statement. Use of this clause causes your query output to be supplied in an XML format, and a number of options are available to allow fairly specific control of exactly how that XML output is styled. I’m going to shy away from the details of this clause for now because XML is a discussion unto itself, and you’ll spend extra time with XML in Chapter 16. So for now, just trust me that it’s better to learn the basics first.
Making Use of Hints Using the OPTION Clause

The `OPTION` clause is a way of overriding some of SQL Server’s ideas of how best to run your query. Because SQL Server really does usually know what’s best for your query, using the `OPTION` clause will more often hurt you than help you. Still, it’s nice to know that it’s there, just in case.

This is another one of those “I’ll get there later” subjects. I’ll talk about query hints extensively when I talk about locking later in the book, but until you understand what you’re affecting with your hints, there is little basis for understanding the `OPTION` clause. As such, I’ll defer discussion of it for now.

The DISTINCT and ALL Predicates

There’s just one more major concept to get through and you’ll be ready to move from the `SELECT` statement on to action statements. It has to do with repeated data.

Let’s say, for example, that you wanted a list of the `ID`S for all of the products that you have sold at least 30 of in an individual sale (more than 30 at one time). You can easily get that information from the `SalesOrderDetail` table with the following query:

```
SELECT ProductID
FROM Sales.SalesOrderDetail
WHERE OrderQty > 30;
```

Code snippet Chap03.sql

What you get back is one row matching the `ProductID` for every row in the `SalesOrderDetail` table that has an order quantity that is more than 30:

```
<table>
<thead>
<tr>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>709</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>715</td>
</tr>
<tr>
<td>863</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>869</td>
</tr>
<tr>
<td>869</td>
</tr>
<tr>
<td>867</td>
</tr>
</tbody>
</table>

(31 row(s) affected)
```

Although this meets your needs from a technical standpoint, it doesn’t really meet your needs from a reality standpoint. Look at all those duplicate rows! While you could look through and see which products sold more than 30 at a time, the number of rows returned and the number of duplicates
can quickly become overwhelming. As with the problems discussed before, SQL has an answer. It comes in the form of the `DISTINCT` predicate on your `SELECT` statement.

Try re-running the query with a slight change:

```sql
SELECT DISTINCT ProductID
FROM Sales.SalesOrderDetail
WHERE OrderQty > 30;
```

Now you come up with a true list of the `ProductID`S that sold more than 30 at one time:

<table>
<thead>
<tr>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>863</td>
</tr>
<tr>
<td>869</td>
</tr>
<tr>
<td>709</td>
</tr>
<tr>
<td>864</td>
</tr>
<tr>
<td>867</td>
</tr>
<tr>
<td>715</td>
</tr>
</tbody>
</table>

(6 row(s) affected)

As you can see, this cut down the size of your list substantially and made the contents of the list more relevant. Another side benefit of this query is that it will actually perform better than the first one. Why? Well, I go into that later in the book when I discuss performance issues further, but for now, suffice it to say that not having to return every single row means that SQL Server may not have to do quite as much work in order to meet the needs of this query.

As the old commercials on television go, “But wait! There’s more!” You’re not done with `DISTINCT` yet. Indeed, the next example is one that you might be able to use as a party trick to impress your programmer friends. You see, this is one that an amazing number of SQL programmers don’t even realize you can do. `DISTINCT` can be used as more than just a predicate for a `SELECT` statement. It can also be used in the expression for an aggregate. What do I mean? Let’s compare three queries.

First, grab a row count for the `SalesOrderDetail` table in AdventureWorks:

```sql
SELECT COUNT(*)
FROM Sales.SalesOrderDetail;
```

If you haven’t modified the `SalesOrderDetail` table, this should yield you around 121,317 rows.

Now run the same query using a specific column to `COUNT`:

```sql
SELECT COUNT(SalesOrderID)
FROM Sales.SalesOrderDetail;
```

Copyright ©2012 John Wiley & Sons, Inc.
Because the SalesOrderID column is part of the key for this table, it can’t contain any NULLs (more on this in Chapter 9). Therefore, the net count for this query is always going to be the same as the COUNT(*) — in this case, it’s 121,317.

**NOTE** Key is a term used to describe a column or combination of columns that can be used to identify a row within a table. There are actually several kinds of keys (you’ll see much more on these in Chapters 6, 8, and 9), but when the word “key” is used by itself, it is usually referring to a table’s primary key. A primary key is a column (or group of columns) that is (are) effectively the unique name for that row. When you refer to a row using its primary key, you can be certain that you will get back only one row, because no two rows are allowed to have the same primary key within the same table.

Now for the fun part. Modify the query again:

```sql
SELECT COUNT(DISTINCT SalesOrderID)
FROM Sales.SalesOrderDetail;
```

Now you get a substantially different result:

```
-----------
31465
(1 row(s) affected)
```

All duplicate rows were eliminated before the aggregation occurred, so you have substantially fewer rows.

**NOTE** Note that you can use DISTINCT with any aggregate function, although I question whether many of the functions have any practical use for it. For example, I can’t imagine why you would want an average of just the DISTINCT rows.

That takes us to the ALL predicate. With one exception, it is a very rare thing indeed to see someone actually including an ALL in a statement. ALL is perhaps best understood as being the opposite of DISTINCT. Where DISTINCT is used to filter out duplicate rows, ALL says to include every row. ALL is the default for any SELECT statement, except for situations where there is a UNION. You will read about the impact of ALL in a UNION situation in the next chapter, but for now, realize that ALL is happening any time you don’t ask for a DISTINCT.
ADDING DATA WITH THE INSERT STATEMENT

By now you should pretty much have the hang of basic SELECT statements. You would be doing well to stop here, save for a pretty major problem. You wouldn’t have very much data to look at if you didn’t have some way of getting it into the database in the first place. That’s where the INSERT statement comes in.

The full syntax for INSERT has several parts:

```
INSERT [TOP ( <expression> ) [PERCENT] ] [INTO] <tabular object>
[<column list>]]
[ OUTPUT <output clause> ]
{ VALUES (<data values>) [(<data values>)] [, _n]}
| <table source>
| EXEC <procedure>
| DEFAULT VALUES
```

This is a bit wordy to worry about now, so let’s simplify it a bit. The more basic syntax for an INSERT statement looks like this:

```
INSERT [INTO] <table>
[<column list>]]
VALUES (<data values>) [(<data values>)] [, _n]
```

Let’s look at the parts.

INSERT is the action statement. It tells SQL Server what it is that you’re going to be doing with this statement; everything that comes after this keyword is merely spelling out the details of that action.

The INTO keyword is pretty much just fluff. Its sole purpose in life is to make the overall statement more readable. It is completely optional, but I highly recommend its use for the very reason that it was added to the statement: it makes things much easier to read. As you go through this section, try a few of the statements both with and without the INTO keyword. It’s a little less typing if you leave it out, but it’s also quite a bit stranger to read. It’s up to you, but settle on one way or the other.

Next comes the table (technically table, view, or a common table expression, but you’re just going to worry about tables for now) into which you are inserting.

Until this point, things have been pretty straightforward. Now comes the part that’s a little more difficult: the column list. An explicit column list (where you specifically state the columns to receive values) is optional, but not supplying one means that you have to be extremely careful. If you don’t provide an explicit column list, each value in your INSERT statement will be assumed to match up with a column in the same ordinal position of the table in order (first value to first column, second value to second column, and so on). Additionally, a value must be supplied for every column, in order, until you reach the last column that both does not accept NULLS and has no default. (You’ll see more about what I mean shortly.) The exception is an IDENTITY column, which should be skipped when supplying values (SQL Server will fill that in for you). In summary, this will be a list of one or more columns that you are going to be providing data for in the next part of the statement.

Finally, you’ll supply the values to be inserted. There are two ways of doing this, but for now, focus on single line inserts that use data that you explicitly provide. To supply the values, you can start
with the VALUES keyword, and then follow that with a list of values separated by commas and enclosed in parentheses. The number of items in the value list must exactly match the number of columns in the column list. The data type of each value must match or be implicitly convertible to the type of the column with which it corresponds (they are taken in order).

**NOTE** On the issue of whether or not to specifically state a value for all columns, I really recommend naming every column every time, even if you just use the DEFAULT keyword or explicitly state NULL. DEFAULT will tell SQL Server to use whatever the default value is for that column (if there isn't one, you'll get an error). What's nice about this is the readability of code; this way it's really clear what you are doing. In addition, I find that explicitly addressing every column leads to fewer bugs. If someone adds a column to that table, an INSERT that lists every column may still work, but one that doesn’t is broken. Worse, if columns are reordered (which is uncommon but possible), you could see data loaded into the wrong fields if you weren't explicit.

Whew! That’s confusing, so let’s practice with this some.

To get started with INSERTS, UPDATES, and DELETES, you’ll want to create a couple of tables to work with. (AdventureWorks is a bit too bulky for just starting out.) To be ready to try these next few examples, you’ll need to run a few statements that you haven't really learned about yet. Try not to worry about the contents of this yet; I’ll get to discussing them fully in Chapter 5.

You can either type these and execute them, or use the file called Chap3CreateExampleTables.sql included with the downloadable files for this book.

**NOTE** This next block of code is what is called a script. This particular script is made up of one batch. You will be examining batches at length in Chapter 11.

```sql
/* This script creates a couple of tables for use with ** several examples in Chapter 3 of Beginning SQL Server ** 2012 Programming */

CREATE TABLE Stores
(
    StoreCode    char(4)      NOT NULL PRIMARY KEY,
    Name         varchar(40)  NOT NULL,
    Address      varchar(40)  NULL,
    City         varchar(20)  NOT NULL,
    State        char(2)      NOT NULL,
    Zip          char(5)      NOT NULL
);

CREATE TABLE Sales
(
    OrderNumber  varchar(20)  NOT NULL PRIMARY KEY,
```
Most of the inserts you’re going to do in this chapter will be to the Stores table you just created, so let’s review the properties for that table. To do this, expand the Tables node of whichever database was current when you ran the preceding script (probably AdventureWorks given the other examples you’ve been running) in the Object Explorer within the Management Studio. Then expand the Columns node, as shown in Figure 3-3. Note that you’re looking for the dbo.Stores table you just created, not the Sales.Store table that came with AdventureWorks.

In this table, every column happens to be a char or varchar.

For your first insert, you can eliminate the optional column list and allow SQL Server to assume you’re providing something for every column:

```
INSERT INTO Stores
VALUES
    ('TEST', 'Test Store', '1234 Anywhere Street', 'Here', 'NY', '00319');
```

As stated earlier, unless you provide a different column list (I’ll cover how to provide a column list shortly), all the values have to be supplied in the same order as the columns are defined in the table. After executing this query, you should see a statement that tells you that one row was affected by your query. Now, just for fun, try running the exact same query a second time. You’ll get the following error:

```
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__Stores__1387E197'.
Cannot insert duplicate key in object 'dbo.Stores'.
The statement has been terminated.
```

**NOTE** The system-generated name you see for the previous primary key constraint will likely differ on your instance of SQL Server. Let this be a reminder to take the extra 10 seconds to put descriptive names on your keys when you create them.
Why did it work the first time and not the second? Because this table has a primary key that does not allow duplicate values for the StoreCode field. If you had changed that one field, you could have left the rest of the columns alone, and it would have taken the new row. You’ll see more of primary keys in the chapters on design and constraints.

So let’s see what you just inserted:

```sql
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

This query yields exactly what you inserted:

<table>
<thead>
<tr>
<th>StoreCode</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>Test Store</td>
<td>1234 Anywhere Street</td>
<td>Here</td>
<td>NY</td>
<td>00319</td>
</tr>
</tbody>
</table>

(1 row(s) affected)

Note that I’ve trimmed a few spaces off the end of each column to help it fit on a page neatly, but the true data is just as you expected it to be.

Now let’s try it again with modifications for inserting into specific columns:

```sql
INSERT INTO Stores
(StoreCode, Name, City, State, Zip)
VALUES
( 'TST2', 'Test Store', 'Here', 'NY', '00319');
```

Note that on the line with the data values this example changed just two things. First, the value you are inserting into the primary key column is different so it won’t generate an error. Second, this query has eliminated the value that was associated with the Address column, because it has omitted that column in the column list. There are a few different instances where you can skip a column in a column list and not provide any data for it in the `INSERT` statement. For now, you’re just taking advantage of the fact that the Address column is not a required column — that is, it accepts NULLs. Because you’re not providing a value for this column and because it has no default (you’ll see more on defaults later on), this column is set to NULL when you perform your `INSERT`. You could verify that by rerunning your test `SELECT` statement with one slight modification:

```sql
SELECT *
FROM Stores
WHERE StoreCode = 'TST2';
```
Now you see something a little different:

<table>
<thead>
<tr>
<th>StoreCode</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST2</td>
<td>Test Store</td>
<td>NULL</td>
<td>Here</td>
<td>NY</td>
<td>00319</td>
</tr>
</tbody>
</table>

(1 row(s) affected)

Notice that a **NULL** was inserted for the column that you skipped.

Note that the columns have to be **nullable** in order to do this. What does that mean? Pretty much what it sounds like: It means that you are allowed to have **NULL** values for that column. Believe me, you will be learning about the nullability of columns at great length in this book, but for now, just realize that some columns allow **NULLs** and some don’t. You can always skip providing information for columns that allow **NULLs**.

If, however, the column is not nullable, one of three conditions must exist, or you will receive an error and the **INSERT** will be rejected:

- The column has been defined with a **default value**. A default is a constant value that is inserted if no other value is provided. You will learn how to define defaults in Chapter 7.

- The column is defined to receive some form of system-generated value. The most common of these is an **IDENTITY** value (covered more in the design chapter), where the system typically starts counting first at one row, increments to two for the second, and so on. These aren’t really row numbers, as rows may be deleted later on and numbers can, under some conditions, get skipped, but they serve to make sure each row has its own identifier. Other less common defaults may include **SYSDATETIME()** or a value retrieved from a **SEQUENCE** (more on the **SEQUENCE** object in Chapter 11).

- You supply a value for the column.

Just for completeness, go ahead and perform one more **INSERT** statement. This time, insert a new sale into the **Sales** table. To view the properties of the **Sales** table, you can either open its Properties dialog box as you did with the **Stores** table, or you can run a system-stored procedure called **sp_help**. **sp_help** reports information about any database object, user-defined data type, or SQL Server data type. The syntax for using **sp_help** is as follows:

```sql
EXEC sp_help <name>
```

To view the properties of the **Sales** table, you just have to type the following into the Query window:

```sql
EXEC sp_help Sales;
```

Available for download on Wrox.com

Code snippet Chap03.sql
Which returns (among other things):

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Type</th>
<th>Computed</th>
<th>Length</th>
<th>Prec</th>
<th>Scale</th>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrderNumber</td>
<td>varchar</td>
<td>no</td>
<td>20</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>StoreCode</td>
<td>char</td>
<td>no</td>
<td>4</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>OrderDate</td>
<td>date</td>
<td>no</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>Quantity</td>
<td>int</td>
<td>no</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>Terms</td>
<td>varchar</td>
<td>no</td>
<td>12</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>TitleID</td>
<td>int</td>
<td>no</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>no</td>
</tr>
</tbody>
</table>

The Sales table has six columns in it, but pay particular attention to the Quantity, OrderDate, and TitleID columns; they are of types that you haven’t done inserts with up to this point.

What you need to pay attention to in this query is how to format the types as you’re inserting them. You do *not* use quotes for numeric values as you have with character data. However, the date data type does require quotes. (Essentially, it goes in as a string and it then gets converted to a date.)

```sql
INSERT INTO Sales
    (StoreCode, OrderNumber, OrderDate, Quantity, Terms, TitleID)
VALUES
    ('TEST', 'TESTORDER', '01/01/1999', 10, 'NET 30', 1234567);
```

This gets back the now familiar *(1 row(s) affected)* message. Notice, however, that I moved around the column order in my insert. The data in the VALUES portion of the insert needs to match the column list, but the column list can be in any order you choose to make it; it does not have to be in the order the columns are listed in the physical table.

**NOTE** Note that while I’ve used the MM/DD/YYYY format that is popular in the United States, you can use a wide variety of other formats (such as the internationally more popular YYYY-MM-DD) with equal success. The default for your server will vary depending on whether you purchase a localized copy of SQL Server or if the setting has been changed on the server.

### Multirow Inserts

Starting with SQL Server 2008, you have the ability to insert multiple rows at one time. To do this, just keep tacking on additional comma-delimited insertion values. For example:

```sql
INSERT INTO Sales
    (StoreCode, OrderNumber, OrderDate, Quantity, Terms, TitleID)
VALUES
    ('TST2', 'TESTORDER2', '01/01/1999', 10, 'NET 30', 1234567),
    ('TST2', 'TESTORDER3', '02/01/1999', 10, 'NET 30', 1234567);
```
This inserts both sets of values using a single statement. To check what you have, go ahead and look in the Sales table:

```sql
SELECT *
FROM Sales;
```

And sure enough, you get back the one row you inserted earlier along with the two rows you just inserted:

<table>
<thead>
<tr>
<th>OrderNumber</th>
<th>StoreCode</th>
<th>OrderDate</th>
<th>Quantity</th>
<th>Terms</th>
<th>TitleID</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESTORDER</td>
<td>TEST</td>
<td>1999-01-01</td>
<td>10</td>
<td>NET</td>
<td>30</td>
</tr>
<tr>
<td>TESTORDER2</td>
<td>TST2</td>
<td>1999-01-01</td>
<td>10</td>
<td>NET</td>
<td>30</td>
</tr>
<tr>
<td>TESTORDER3</td>
<td>TST2</td>
<td>1999-02-01</td>
<td>10</td>
<td>NET</td>
<td>30</td>
</tr>
</tbody>
</table>

(3 row(s) affected)

**NOTE** This feature has the potential to really boost performance in situations where you are performing multiple `INSERT`s. Previously, your client application would have to issue a completely separate `INSERT` statement for each row of data you wanted to insert (there were some ways around this, but they required extra thought and effort that it seems few developers were willing to put in). Using this method can eliminate many round trips to your server; just keep in mind that it also means that your application will not be backward compatible with prior versions of SQL Server.

### The INSERT INTO . . . SELECT Statement

What if you have a block of data that you want to `INSERT`? As you have just seen, you can perform multirow `INSERT`s explicitly, but what if you want to `INSERT` a block of data that can be selected from another source, such as:

- Another table in your database
- A totally different database on the same server
- A heterogeneous query from another SQL Server or other data
- The same table (usually you’re doing some sort of math or other adjustment in your `SELECT` statement, in this case)

The `INSERT INTO . . . SELECT` statement can `INSERT` data from any of these. The syntax for this statement comes from a combination of the two statements you’ve seen thus far — the `INSERT` statement and the `SELECT` statement. It looks something like this:

```sql
INSERT INTO <table name>
[<column list>]
<SELECT statement>
```
Adding Data with the INSERT Statement

The result set created from the SELECT statement becomes the data that is added in your INSERT statement.

Let’s check this out by doing something that, if you get into advanced coding, you’ll find yourself doing all too often — SELECTing some data into some form of temporary table. In this case, I’ll show you how to declare a variable of type table and fill it with rows of data from your Orders table:

**NOTE** Like the prior database-creation example, this next block of code is what is called a script. Again, you will be examining scripts and batches at length in Chapter 11.

```sql
/* This next statement is going to use code to ** change the "current" database to AdventureWorks. ** This makes certain, right in the code, that you are going ** to the correct database. */
USE AdventureWorks;

/* This next statement declares your working table. ** This particular table is actually a variable you are declaring ** on the fly. */
DECLARE @MyTable Table
(
    SalesOrderID      int,
    CustomerID        char(5)
);

/* Now that you have your table variable, you're ready ** to populate it with data from your SELECT statement. ** Note that you could just as easily insert the ** data into a permanent table (instead of a table variable). */
INSERT INTO @MyTable
    SELECT SalesOrderID, CustomerID
    FROM AdventureWorks.Sales.SalesOrderHeader
    WHERE SalesOrderID BETWEEN 44000 AND 44010;

-- Finally, make sure that the data was inserted like you think
SELECT *
FROM @MyTable;
```

Code snippet Chap03.sql

Available for download on Wrox.com

Copyright ©2012 John Wiley & Sons, Inc.
This should yield results that look like this:

```
(11 row(s) affected)
SalesOrderID  CustomerID
-----------  ----------
 44000       27918
 44001       28044
 44002       14572
 44003       19325
 44004       28061
 44005       26629
 44006       16744
 44007       25555
 44008       27678
 44009       27759
 44010       13680
```

(11 row(s) affected)

The first instance of `(11 row(s) affected)` you see is the effect of the `INSERT...SELECT` statement at work. Your `SELECT` statement returned 11 rows, so that’s what got `INSERT`ed into your table. You then used a straight `SELECT` statement to verify the `INSERT`.

**NOTE**

Note that if you try running a `SELECT` against `@MyTable` by itself (that is, outside this script), you’re going to get an error. `@MyTable` is a declared variable, and it exists only as long as it remains in scope within the running batch. After that, it is automatically destroyed.

It’s also worth noting that you could have used what’s called a temporary table. This is similar in nature, but it doesn’t work in quite the same way. You’ll discover a lot more about temp tables and table variables in Chapters 11 through 13.

## CHANGING WHAT YOU’VE GOT WITH THE UPDATE STATEMENT

The `UPDATE` statement, like most SQL statements, does pretty much what it sounds like it does — it updates existing data. The structure is a little bit different from a `SELECT`, although you’ll notice definite similarities. Like the `INSERT` statement, it has a fairly complex set of options, but a more basic version that will meet the vast majority of your needs.

The full syntax supports the same `TOP` and similar predicates that were supported under `SELECT` and `INSERT`:

```
UPDATE [TOP ( <expression> ) [PERCENT] ] <tabular object>
    SET <column> = <value>[.WRITE(<expression>, <offset>, <length>)]
        [,<column> = <value>[.WRITE(<expression>, <offset>, <length>)]
          [ OUTPUT <output clause> ]
    [ FROM <source table(s)>]
    [WHERE <restrictive condition>]
```

Copyright ©2012 John Wiley & Sons, Inc.
Let's look at the more basic syntax:

```sql
UPDATE <table name>
SET <column> = <value> [,<column> = <value>] 
[FROM <source table(s)>]
[WHERE <restrictive condition>]
```

An `UPDATE` can be created from multiple tables, but can affect only one table. What do I mean by that? Well, you can build a condition, or retrieve values from any number of different tables, but only one table at a time can be the subject of the `UPDATE` action. Don’t sweat this one too much. I haven’t even started talking about joining multiple tables yet (next chapter folks!), so I won’t get into complex `UPDATE` statements here. For now, you’ll look at simple updates.

Let’s start by doing some updates to the data that you inserted in the `INSERT` statement section. Try to rerun that query to look at one row of inserted data. (Don’t forget to switch back to the `pubs` database.)

```sql
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

This returns the following:

<table>
<thead>
<tr>
<th>StoreCode</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>Test Store</td>
<td>1234 Anywhere Street</td>
<td>Here</td>
<td>NY</td>
<td>00319</td>
</tr>
</tbody>
</table>

Here’s how to update the value in the `City` column:

```sql
UPDATE Stores
SET City = 'There'
WHERE StoreCode = 'TEST';
```

Much like when you ran the `INSERT` statement, you don’t get much back from SQL Server:

```
(1 row(s) affected)
```

Yet when you run your `SELECT` statement again, you see that the value has indeed changed:

<table>
<thead>
<tr>
<th>StoreCode</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>Test Store</td>
<td>1234 Anywhere Street</td>
<td>There</td>
<td>NY</td>
<td>00319</td>
</tr>
</tbody>
</table>
Note that you could have changed more than one column just by adding a comma and the additional column expression. For example, the following statement would have updated both columns:

```
UPDATE Stores
SET City = 'There', State = 'NY'
WHERE StoreCode = 'TEST';
```

If you choose, you can use an expression for the `SET` clause instead of the explicit values you’ve used thus far. For example, you could add a suffix on all of the store names by concatenating the existing store name with a suffix:

```
UPDATE Stores
SET Name = Name + ' - ' + StoreCode;
```

After executing that `UPDATE`, run a `SELECT` statement on `Stores`:

```
SELECT *
FROM Stores
```

You should see the `Name` suffixed by the `StoreCode`:

<table>
<thead>
<tr>
<th>StoreCode</th>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>Test Store - TEST</td>
<td>1234 Anywhere Street</td>
<td>There</td>
<td>NY</td>
<td>00319</td>
</tr>
<tr>
<td>TST2</td>
<td>Test Store - TST2</td>
<td>NULL</td>
<td>Here</td>
<td>NY</td>
<td>00319</td>
</tr>
</tbody>
</table>

As you can see, a single `UPDATE` statement can be fairly powerful. Even so, this is really just the beginning. You’ll see even more advanced updates in later chapters.

**NOTE** While SQL Server is nice enough to let you `UPDATE` pretty much any column (there are a few that we can’t, such as `timestamp`), be very careful about updating primary keys. Doing so puts you at very high risk of orphaning other data (data that has a reference to the data you’re changing).

For example, the `StoreCode` field in the `Stores` table is a primary key. If you decide to change `StoreCode 10 to 35 in Stores, any data in the Sales table that relates to that store may be orphaned and lost if the `StoreCode` value in all of the records relating to `StoreCode 10 is not also updated to 35. As it happens, there is a constraint that references the `Stores` table, so SQL Server would prevent such an orphaning situation in this case (you’ll investigate constraints in Chapter 7), but updating primary keys is risky at best.
THE DELETE STATEMENT

The version of the DELETE statement that’s covered in this chapter may be one of the easiest statements of them all. Even in the more complex syntax, there’s no column list, just a table name and (usually) a WHERE clause. The full version looks like this:

```sql
DELETE [TOP ( <expression> ) [PERCENT] ] [FROM] <tabular object>
[ OUTPUT <output clause> ]
[FROM <table or join condition>]
[WHERE <search condition> | CURRENT OF [GLOBAL] <cursor name>]
```

The basic syntax couldn’t be much easier:

```sql
DELETE <table name>
[WHERE <condition>]
```

The WHERE clause works just like all the WHERE clauses you’ve seen thus far. You don’t need to provide a column list because you are deleting the entire row. (You can’t DELETE half a row, for example; that would be an UPDATE.)

Because this is so easy, you can master this by performing only a couple of quick DELETEs that are focused on cleaning up the INSERTs that you performed earlier in the chapter. First, run a SELECT to make sure the first of those rows is still there:

```sql
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

If you haven’t already deleted it, you should come up with a single row that matches what you added with your original INSERT statement. Now let’s get rid of it:

```sql
DELETE Stores
WHERE StoreCode = 'TEST';
```

Note that you’ve run into a situation where SQL Server is refusing to DELETE this row because of referential integrity violations:

```
Msg 547, Level 16, State 0, Line 1
The DELETE statement conflicted with the REFERENCE constraint
"FK__Sales__StoreCode__1B29035F". The conflict occurred in
database "AdventureWorks", table "dbo.Sales", column 'StoreCode'.
The statement has been terminated.
```

SQL Server won’t let you delete a row if it is referenced as part of a foreign key constraint. You’ll see much more on foreign keys in Chapter 7, but for now, just keep in mind that if one row references
another row (either in the same or in a different table — it doesn’t matter) using a foreign key, the referencing row must be deleted before the referenced row can be deleted. One of your `INSERT` statements inserted a record into the `Sales` table that had a `StoreCode` of `TEST` — this record is referencing the record you have just attempted to delete.

Before you can delete the record from the `Stores` table, you must delete the record it is referencing in the `Sales` table:

```sql
DELETE Sales
WHERE StoreCode = 'TEST';
```

Now you can successfully rerun the first `DELETE` statement:

```sql
DELETE Stores
WHERE StoreCode = 'TEST';
```

You can do two quick checks to verify that the data was indeed deleted. The first happens automatically when the `DELETE` statement is executed; you should get a message telling you that one row was affected. The other quick check is to rerun the `SELECT` statement; you should get zero rows back.

For one more easy practice `DELETE`, you can also kill that second row by making just a slight change:

```sql
DELETE Sales
WHERE StoreCode = 'TST2';
```

That’s it for simple `DELETEs`! As with the other statements in this chapter, you’ll come back to the `DELETE` statement when you learn about more complex search conditions.

**SUMMARY**

T-SQL is SQL Server’s own brand of ANSI/ISO SQL or Structured Query Language. T-SQL is largely ANSI/ISO compliant, but it also has a number of its own extensions to the language. You’ll see more of those in later chapters.

Even though, for backward compatibility, SQL Server has a number of different syntax choices that are effectively the same, wherever possible you ought to use the ANSI form. Where there are different choices available, I will usually show you all of the choices, but again, stick with the ANSI/ISO version wherever possible. This is particularly important for situations where you think your backend — or database server — might change at some point. Your ANSI code will more than likely run on the new database server; however, code that is only T-SQL definitely will not.
In this chapter, you have gained a solid taste of making use of single table statements in T-SQL, but the reality is that you often need information from more than one table. In the next chapter, you will learn how to make use of JOINs to allow you to use multiple tables.

**EXERCISES**

1. Write a query that outputs all of the columns and all of the rows from the `Product` table (in the `Production` schema) of the AdventureWorks database.

2. Modify the query in Exercise 1 so it filters down the result to just the products that have no `ProductSubcategoryID`. (HINT: There are 209, and you will need to be looking for NULL values.)

3. Add a new row into the `Location` (in the `Production` schema) table in the AdventureWorks database.

4. Remove the row you just added.
## WHAT YOU LEARNED IN THIS CHAPTER

<table>
<thead>
<tr>
<th>TOPIC</th>
<th>CONCEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>The most fundamental of all SQL DML (Data Manipulation Language) statements, <strong>SELECT</strong> is used to retrieve data from a table. You will commonly <strong>SELECT</strong> fields FROM tables WHERE some conditions are met.</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>You may aggregate data in your <strong>SELECT</strong> statements, rather than simply return the rows exactly as they appear in the table, by using aggregators such as <strong>SUM</strong>, <strong>MAX</strong>, <strong>MIN</strong>, and <strong>AVG</strong> on the fields you want to aggregate and adding a <strong>GROUP BY</strong> clause to group by the other fields.</td>
</tr>
<tr>
<td>Filtering</td>
<td>Filtering with <strong>WHERE</strong> occurs before aggregation, and filtering with <strong>HAVING</strong> happens after. Filtering is done with Boolean tests.</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>To remove duplicate rows from your results, you may use <strong>SELECT DISTINCT</strong>. Duplicates are checked across all fields returned.</td>
</tr>
<tr>
<td>INSERT</td>
<td>To put data into a table, <strong>INSERT</strong> it. <strong>INSERT</strong> takes a list of the fields you want to insert into, the table you want to load the data into, and either a list of literal values using <strong>VALUES</strong>, or a <strong>SELECT</strong> statement that produces a compatible field list.</td>
</tr>
<tr>
<td>UPDATE</td>
<td>To alter data in a table, you use <strong>UPDATE table_name</strong>. Specify which values you want to change with <strong>SET field_name = value</strong>, and add a <strong>WHERE</strong> condition to limit the table rows updated.</td>
</tr>
<tr>
<td>DELETE</td>
<td>You can remove rows from your tables (permanently) with <strong>DELETE FROM table_name</strong>. Use a <strong>WHERE</strong> clause to limit the rows deleted.</td>
</tr>
</tbody>
</table>