CHAPTER 4

SQL Statement Primer

In This Chapter

In this chapter, I'm going to show you some of the key SQL statements that you will be using when you develop your applications. These statements will allow you to create tables, views, and indexes. The rest of the statements can be used to add, remove, change, and retrieve rows from your database.

Using SQL Statements

A detailed knowledge of SQL isn't necessary for most programmers. However, it will be impossible to write a database program without knowing a little bit about the language. The statements I'm going to cover in this chapter apply to all of the database systems that will be discussed in this book.

Note

SQL for Dummies: If you really want to learn more about the SQL language, read the book SQL for Dummies, 3rd Edition, by Allen G. Taylor. This is a good introduction to the SQL language and covers all of the essential elements of the language. More advanced users should refer to the database vendor's documentation for their extensions to the SQL language.

SQL statements

The SQL language consists of a series of statements that perform specific tasks (see Table 4-1). There are statements to create databases and tables, statements to add and delete rows in a table, and statements to retrieve rows from a table or set of tables. There are other statements that deal with data security and data integrity. These statements are constructed according to a set of complex rules that vary slightly from one database system to another. However, for most users, these differences aren't all that important.

Table 4-1
Some Common SQL Statements

Statement Name	***Description***
`Create Index`	Builds an index on a set of columns on a table.
`Create Table`	Builds an empty table in a database.
`Create View`	Builds a view.
`Delete`	Removes rows from a table.
`Drop Index`	Deletes an index from a database.
`Drop Table`	Deletes a table from a database.
`Drop View`	Removes a view from a database.
`Insert`	Adds rows to a table.
`Select`	Retrieves rows from a table.
`Update`	Changes the data values of one or more columns in the table.

Data definition language statements

The **Create Table**, **Drop Table, Create View**, **Drop View Create Index**, and **Drop Index** statements are known as *Data Definition Language* (DDL) statements, while the **Insert**, **Delete**, **Update**, and **Select** statements are known as *Data Manipulation Language* (DML) statements. In most database systems today, you rarely execute DDL statements when you want to create a database structure. Instead, you use a utility supplied with the database system that allows you to fill in all of the information into a table, or you use a wizard that will help you create your table or index.

This doesn't mean that the DDL statements aren't used. It merely means that you enter the information in a different fashion. The database utility usually includes a feature that will allow you to generate the SQL statements from the definitions you entered. Then you might use these SQL statements to create a copy of the database on another computer or include them in your application if you want your users to be able to create the database structures on the fly.

SQL data types

Each column in a table must have a data type associated with it. The data type you choose for a column must be compatible with a Visual Basic variable data type. Table 4-2 lists some of the most common data types used by SQL, along with their equivalent data types in Visual Basic.

Table 4-2
Some Common SQL Data Types

SQL Data Type	*Visual Basic Data Type*	*Description*
Char	String	A fixed-length string of characters.
Date	Date	A value containing a date and time value. (Available with Oracle only.)
Datetime	Date	A value containing a date and time value. (Available with SQL Server only.)
Decimal	Currency	An exact numeric value of the specified size.
Float	Double	A 64-bit floating-point number.
Int	Long	A 32-bit integer.
Money	Currency	An exact numeric value. (Available with SQL Server only).
Number	Currency	An exact numeric value. (Available with Oracle only.)
Real	Single	A 32-bit floating-point number.
Smallint	Integer	A 16-bit integer.
Varchar	String	Variable-length character string.

These data types can be loosely grouped into four main types: exact numeric values, floating point values, string values, and date values. Most database servers also offer many other data types to choose from.

Cross-Reference

For more detailed information about the data types available in a particular database, see Chapter 23, "Overview of SQL Server," Chapter 26, "Overview of Oracle 8*i*," or Chapter 29, "Overview of Microsoft Jet."

Exact numeric data types

Exact numeric data types represent numbers by using an exact value. These data types generally fall into two sub classes: *integer values* and *packed decimal values.* Integer values store their numbers as a binary value. This offers more efficient storage for large numbers than when you use a packed decimal value.

Tip

Money, money, money: If you need to perform calculations using currency values, you should always use an exact numeric data type.

Packed decimal values store numbers as a string of numeric digits. Four bits are used to represent a value from zero to nine. Most database servers will allow you to determine the number of digits you want when you specify the data type.

The advantage of exact numeric values is that when you perform arithmetic with them, you never lose accuracy. This isn't true with floating point values.

Floating point data types

Floating point data types represent numbers by breaking them into two pieces: a *mantissa* and an *exponent*. Floating point numbers are expressed in terms of a value time 10 to some power. For instance, the value 12,345 is written as 1.234×10^5 and often displayed on the computer as 1.2345E5, where E means 10 raised to this poser. In this example, the mantissa is 1.2345 and the exponent is 5.

Because of the way they are stored, floating point numbers are only accurate to so many decimal places. This allows you to represent very large numbers with much less storage than what would be required if you stored every single digit. Generally, *single values* have about five decimal places of accuracy, while *double values* are accurate to about ten decimal places.

Tip

Maybe, maybe not: You'll probably never need to use a floating point data type in your database, because most people don't appreciate adding 1,000,000.02 + 0.01 and getting 1,000,000 because the floating point value stored only 5 digits of information in the mantissa.

String data types

String data types hold character information. There are two different types of character strings: *fixed-length* and *variable-length*. The fixed-length strings always reserve the same amount of space in a table whether you store one character in the column or fifty. Variable-length strings, on the other hand, store only the characters you have in the string, plus some additional information that holds the length of the string.

In general, you should choose variable-length strings over fixed-length strings. This tends to save space in your database, especially if the amount of data you store in the column varies significantly from one row to the next. Using variable-length strings also allows you to create your strings with a larger maximum size. This is useful in situations where you may have an unusually large value that you don't want to truncate, such as a person's name.

Fixed-length strings are good when the size of each value remains relatively constant, as with a two-character abbreviation for a state or a product identifier code. This is especially true for small strings where the extra overhead to keep track of the true length of the string occupies more space then the string itself.

Date data types

Date data types are almost always unique to a particular database system. Even though there isn't much compatibility among the database vendors' implementations, the alternatives are worse. You could allocate an eight-character string and use the first four characters for the year, the next two for the month, and the last two for the year. You could also use an integer value to track the number of days since 1900 or since the day your organization was created.

Both of these methods have a drawback: the lack of integrated support for the values by the database server. If you store your date values as a character string instead of a date data type, when you use an interactive query tool, all you'll see is the raw, unformatted value. When using these values with Visual Basic, you'll have to manually convert your values to and from a **Date** variable in order to take advantage of the wide range of date and time functions already included in Visual Basic. In the long run, using the supplied date data types from the database server is a much better idea.

Testing SQL statements

One of the advantages of SQL is that the same language can be used interactively or embedded in your application. This means that you can code and test your SQL statements using an interactive query tool and then add them to your program. While the query tools differ depending on the database system you're using, they all do the same thing. You enter your statement and click on a button to execute it. The results will then be displayed on your computer.

For the examples in this chapter, I'm going to use the SQL Server database and the Query Analyzer tool to run the queries against the book's sample database. However, once you create and load the sample database in your database system of choice, you will be able to use the corresponding query tool to run the same examples.

The Select statement

Of the statements in the SQL language, the **Select** statement is the most commonly used. Its purpose is to identify the rows you want to retrieve from the database.

Here is the syntax for this statement:

```
Select [<selectoption>]<selectexpression>
[,<selectexpression>]...
From <tableref> [, <tableref>] ...
[Where <expression>]
[Order By <expression> [Asc|Desc]
   [,<expression> [Asc|Desc] ]...
```

where

```
<selectoption> ::= All | Distinct | Top <number>
<selectexpression> ::=  * | <selectitem> [ [As] <alias> ]
<selectitem> ::= <column> | <table>.<column> |
   <function> ( [Distinct]<expression> ) |  <expression>
<function> ::= Count | Max | Min | Sum
```

and

```
<alias> is an alternate name of a column or table.
<expression> is a valid expression.
<number> is a valid number.
```

The **Select** statement is the most complicated statement in SQL. The above syntax represents only a small part of the full **Select** statement syntax. However, you will rarely need anything beyond these clauses when building your application. A **Select** statement is composed of a series of clauses, such as **From**, **Where**, and so on. Only the **From** clause is required. I'm going to discuss how the basic Select statement works, and then discuss each of the clauses that work with it.

Simple Select statements

To use a **Select** statement, all you need to do is identify the table and the columns you want to retrieve from the database. Immediately following the **Select** keyword is the list of columns you want to retrieve, and the **From** clause specifies the name of the table you want to access.

Retrieving all columns

The following statement retrieves all of the columns from the Customers table in the sample database you'll find in the CD-ROM:

```
Select *
From Customers
```

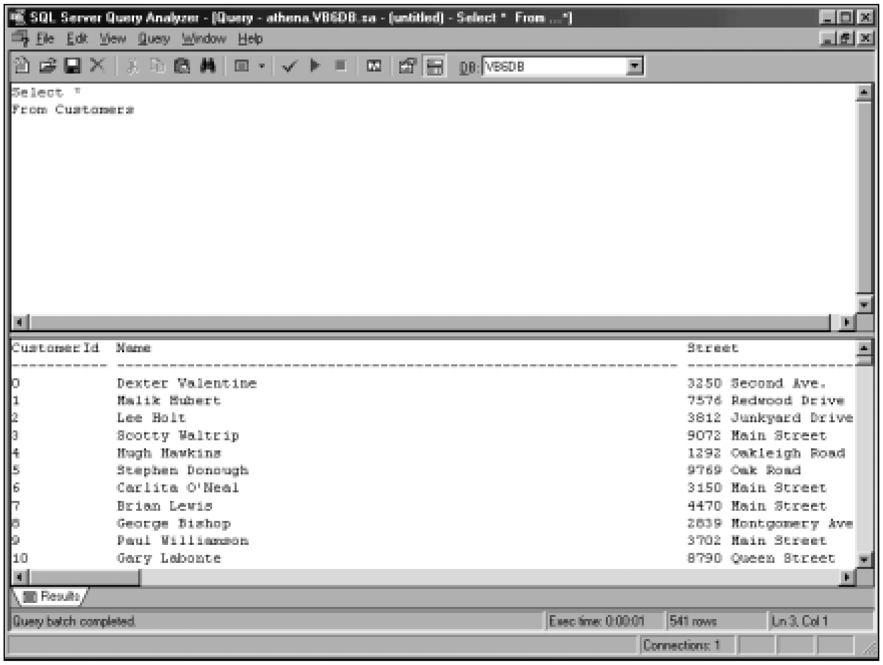
The asterisk (*) implies that you want to retrieve every column from the table. Running this statement in Query Analyzer should generate results similar to those shown in Figure 4-1.

Retrieving a list of columns

If you only need a few specific columns, then you should replace the asterisk with the list of column names you want returned, as shown here:

```
Select CustomerId, Name, Zip
From Customers
```

Each column name should be separated from the previous column by a comma (see Figure 4-2).

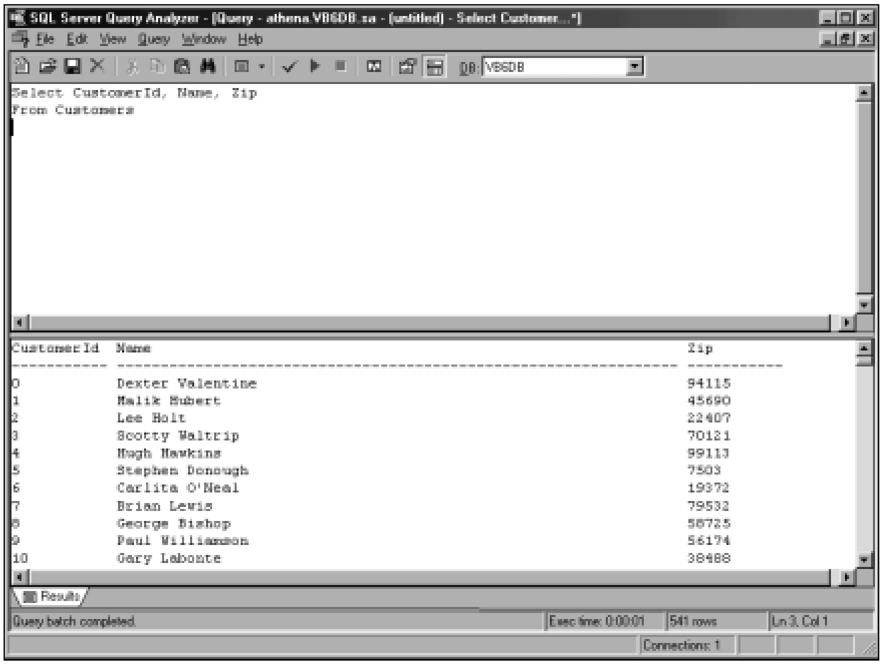

SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Select * From ...*]
File Edit View Query Window Help
DB: VB6DB
Results
Exec time: 0:00:01
541 rows
Ln 3, Col 1
Connections: 1

```
Select *
From Customers
```

```
CustomerId  Name                                                                  Street
----------- --------------------------------------------------------------------- ------------------
0           Dexter Valentine                                                      3250 Second Ave.
1           Malik Hubert                                                          7576 Redwood Drive
2           Lee Holt                                                              3812 Junkyard Drive
3           Scotty Waltrip                                                        9072 Main Street
4           Hugh Hawkins                                                          1292 Oakleigh Road
5           Stephen Donough                                                       9769 Oak Road
6           Carlita O'Neal                                                        3150 Main Street
7           Brian Lewis                                                           4470 Main Street
8           George Bishop                                                         2839 Montgomery Ave
9           Paul Williamson                                                       3702 Main Street
10          Gary Labonte                                                          8790 Queen Street
```

```
Query batch completed.
```

Figure 4-1: Running a simple query to retrieve all columns from the Customer table


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Select Customer...*]
File Edit View Query Window Help
DB: VB6DB
Results
Exec time: 0:00:01
541 rows
Ln 3, Col 1
Connections: 1

```
Select CustomerId, Name, Zip
From Customers
```

```
CustomerId  Name                                                                  Zip
----------- --------------------------------------------------------------------- -----------
0           Dexter Valentine                                                      94115
1           Malik Hubert                                                          45690
2           Lee Holt                                                              22407
3           Scotty Waltrip                                                        70121
4           Hugh Hawkins                                                          99113
5           Stephen Donough                                                       7503
6           Carlita O'Neal                                                        19372
7           Brian Lewis                                                           79532
8           George Bishop                                                         58725
9           Paul Williamson                                                       56174
10          Gary Labonte                                                          38488
```

```
Query batch completed.
```

Figure 4-2: Running a simple query with a list of columns

Selecting a subset of a table

Returning an entire table is not terribly useful in an application program. Typically, you will want to retrieve only a single row or a handful of rows that are related to some other value. This is where the **Where** clause comes into play. The **Where** clause allows you to specify a search expression that identifies the set of rows you want to return.

Note

Where oh where is my favorite clause: If the **Select** statement is the most commonly used statement in the SQL language, the **Where** clause is the most commonly used clause. It is used in a number of other statements, including the **Delete** and **Update** statements.

Using simple search expressions

The trick to using a **Where** clause is to create a search expression that will only return the row or rows you want. For instance, let's assume that you want all of the information about a customer 431. The information is stored in the Customers table. Thus, the search expression `CustomerId = 431` would retrieve all of this information. Since CustomerId column is the primary key for the Customers table, only a single row will be returned by the following **Select** statement (see Figure 4-3):

```
Select *
From Customers
Where CustomerId = 431
```

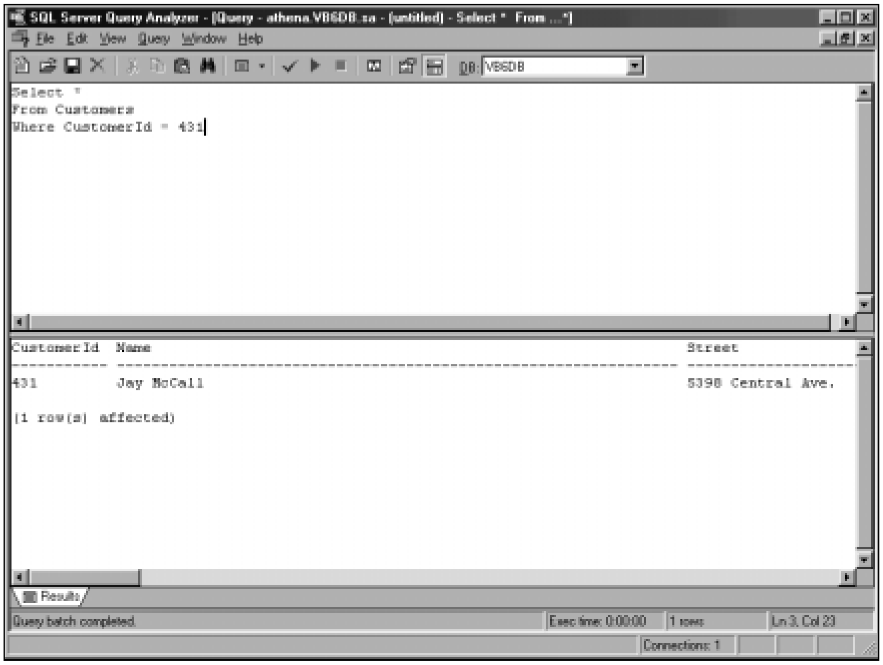
Note

Searching for expressions with all the wrong operators: SQL supports all of the same operators that Visual Basic includes (=, <, >, <=, >=, <>, **Not**, **And**, and **Or**) to make it easy to build an expression. SQL also supports a few other operators, such as **In** (discussed in Nested Queries below) and **Like,** which is used to match a specified pattern, and which may include wild card characters. Of course, parentheses may also be used to ensure that the expression is evaluated properly.

Only those rows containing a CustomerId value of 431 will be returned. Since CustomerId is the primary key of this table, you know that each value of CustomerId is unique, so at most, one row will be returned. Note that if you specify a value for CustomerId that isn't in the table, no rows will be returned.

Of course, if you use an expression that is true for multiple rows, then multiple rows will be returned. The following **Select** statement may return multiple rows from the Customers table, since there may be multiple rows where the State column contains the value "MD" (see Figure 4-4):

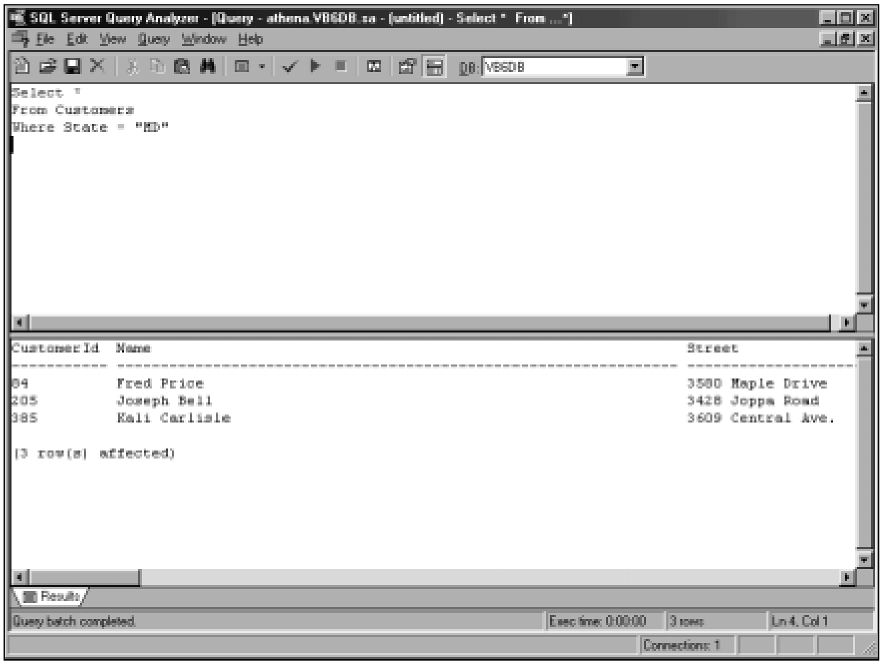
```
Select *
From Customers
Where State = "MD"
```


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Select * From ...*]
File Edit View Query Window Help
DB: VB6DB
Results
Query batch completed.
Exec time: 0:00:00
1 rows
Ln 3, Col 23
Connections: 1

```
Select *
From Customers
Where CustomerId = 431
```

```
CustomerId  Name                                                                 Street
----------- -------------------------------------------------------------------- -------------------
431         Jay McCall                                                           5398 Central Ave.

(1 row(s) affected)
```

Figure 4-3: Selecting information about CustomerId 431


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Select * From ...*]
File Edit View Query Window Help
DB: VB6DB
Results
Query batch completed.
Exec time: 0:00:00
3 rows
Ln 4, Col 1
Connections: 1

```
Select *
From Customers
Where State = "MD"
```

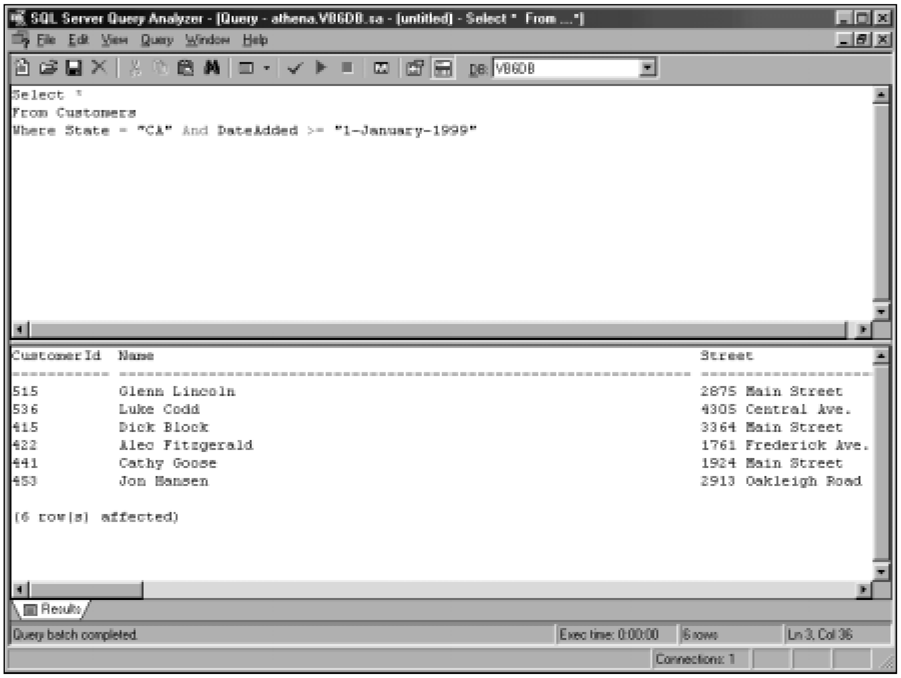
```
CustomerId  Name                                                                 Street
----------- -------------------------------------------------------------------- -------------------
84          Fred Price                                                           3580 Maple Drive
205         Joseph Bell                                                          3428 Joppa Road
385         Kali Carlisle                                                        3609 Central Ave.

(3 row(s) affected)
```

Figure 4-4: Selecting customers from Maryland

More complex search expressions

Search expressions can be as complicated as you want. You can use **And**, **Or,** and **Not** to compile multiple simple expressions together to narrow the search. The following **Select** statement returns all of the customers who were added to the database since 1999 and who also live in California (see Figure 4-5):

```
Select *
From Customers
Where State = "CA" And DateAdded >= "1-January-1999"
```


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Select * From ...*]
File Edit View Query Window Help
DB: VB6DB
Results

```
Select *
From Customers
Where State = "CA" And DateAdded >= "1-January-1999"
```

```
CustomerId  Name                                                               Street
----------- ------------------------------------------------------------------ ------------------
515         Glenn Lincoln                                                      2875 Main Street
536         Luke Codd                                                          4305 Central Ave.
415         Dick Block                                                         3364 Main Street
422         Alec Fitzgerald                                                    1761 Frederick Ave.
441         Cathy Goose                                                        1924 Main Street
453         Jon Hansen                                                         2913 Oakleigh Road

(6 row(s) affected)
```

Query batch completed. | Exec time: 0:00:00 | 6 rows | Ln 3, Col 36
Connections: 1

Figure 4-5: Retrieving all the customers living in California who were added since 1999

Note

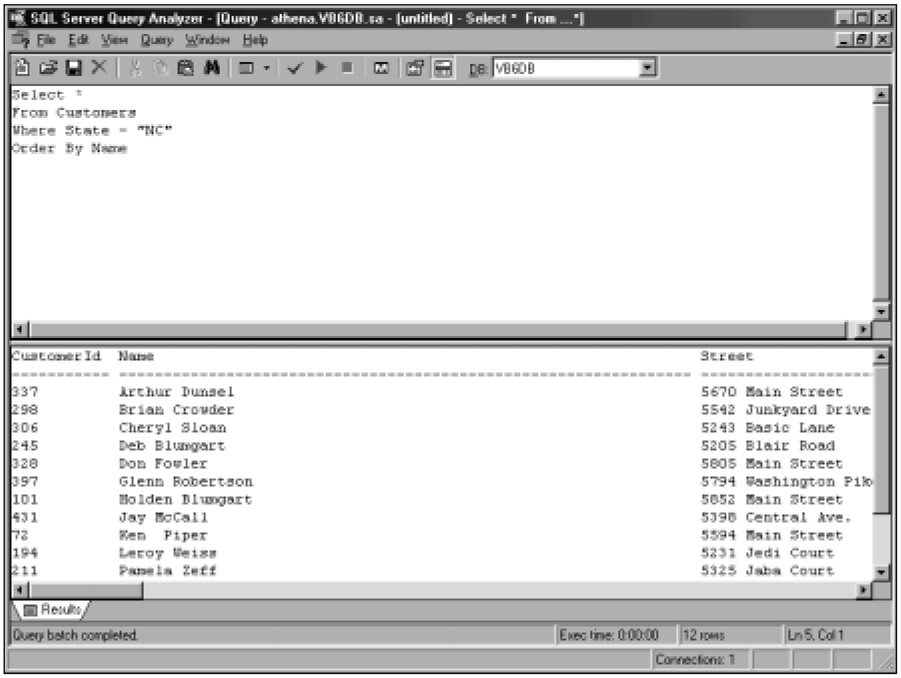
Waiting for it to end: Always try to include at least one column in your search expression that is part of an index. Otherwise, the database server will have to search through every row in the table to find the rows you want. While searching the whole table can be fairly quick for small tables, it can take a long time for large tables.

Sorting results

By default, the **Select** statement doesn't return rows in any particular order. In many cases this isn't a problem, but if you want to display these rows to the user, you might find it beneficial to sort them before they're displayed with the **Order By** clause.

Order By follows the **Where** clause and includes the list of columns that you want to use to sort the results. If you follow a column name with the key **Asc** or **Desc**, that particular column will be sorted in ascending or descending order, respectively. If you don't specify either keyword, the data will be sorted in ascending order. In the statement below, I'm going to retrieve all of the customers who live in North Carolina and sort them by their name (see Figure 4-6):

```
Select *
From Customers
Where State = "NC"
Order By Name
```


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Select * From ...*]
File Edit View Query Window Help
DB: VB6DB

```
Select *
From Customers
Where State = "NC"
Order By Name
```

```
CustomerId  Name                                                              Street
----------- ----------------------------------------------------------------- --------------------
337         Arthur Dunsel                                                     5670 Main Street
298         Brian Crowder                                                     5542 Junkyard Drive
306         Cheryl Sloan                                                      5243 Basic Lane
245         Deb Blumgart                                                      5205 Blair Road
328         Don Fowler                                                        5805 Main Street
397         Glenn Robertson                                                   5794 Washington Pik
101         Holden Blumgart                                                   5852 Main Street
431         Jay McCall                                                        5398 Central Ave.
72          Ken  Piper                                                        5594 Main Street
194         Leroy Weiss                                                       5231 Jedi Court
211         Pamela Zeff                                                       5325 Jaba Court
```

Figure 4-6: Sorting rows retrieved from a table

Note that since the data in the Name field is stored first name then last name, the results are sorted by the person's first name.

Cross-Reference

For more information about foreign keys and keys in general, refer to Chapter 2, "Indexes and Keys."

Using multiple tables

The **Select** statement allows you to combine information from multiple tables into a single "virtual table." This "virtual table" can't be updated, but it makes it easier when you need to collect information you want to display in your application.

Note

Join operations: The technical term for combining the rows and columns in two or more tables is known as a *join operation.*

The wrong way to use two tables

The **Select** statement allows you to specify a list of tables in the **From** clause. However, the results are probably not what you would expect. Consider the following tables. Each table has three rows, with two columns in each row. Each letter represents a specific value in a particular column.

```
Table A: {{A, I}, {B, J}, {C, K}}
Table B: {{X, I}, {Y, J}, {Z, K}}
```

If you specify two tables in the **From** clause, the **Select** perform would look like this and you'll get the following result:

```
Select *
From A, B
```

Note that the **Select** operation matched every row in the first table with each row in the second. This created a table with nine rows, each row having four columns. While there may be cases where you want this result, I can't think of any off the top of my head.

```
{{A, I, X, I}, {A, I, Y, J}, {A, I, Z, K},
{B, J, X, I}, {B, J, Y, J}, {B, J, Z, K},
{C, K, X, I), {C, K, Y, J}, {C, K, Z, K}}
```

The right way to use two tables

```
Generally when you want to use two tables, it is because the
two tables are related to each other. This means that the
tables have one or more columns in common. These columns could
be part of a foreign key relationship. Suppose that Table A and
```

```
Table B have the Column2 in common, which is the second column
in each table. Then the following Select statement would allow
you to join the two tables together based on the rows that have
a common value in their second column:

Select *
From A, B
Where A.Column2 = B.Column2
```

This **Select** statement would then generate the following result:

```
{{A, I, X, I}, {B, J, Y, J}, {C, K, Z, K}}
```

Note that even though Column2 values are identical, they are repeated twice because the rows were appended to each other. Also, if you look back at the previous set of results, you will find these three rows buried. The **Where** clause merely filtered out the rows where the values in Column2 didn't match.

Note

Equijoins: A join that uses the **Where** clause to match column values in different tables is known as an *equijoin,* which is short for equality join.

Resolving column names

In the above example, I had two tables with the same column name. In order to know which column is associated with which table, it is necessary to qualify the column name by using the table name, as shown in the example below:

```
Select LastName, StateName
From Customers, States
Where Customers.State = States.State
```

To make life easier, you may want to use *table aliases,* which allow you to define an alternate name for your table. The table aliases are specified in the **From** clause by following the table name with the alternate name you want to use for the table. Personally, I prefer to use short one- or two-character abbreviations for table aliases, but you can choose whatever size name you want. Using table aliases, I can rewrite the previous query as follows:

```
Select LastName, StateName
From Customers C, States S
Where C.State = S.State
```

Note that using table aliases can shorten the expression in the **Where** clause. While this doesn't save much in this particular example, it can make a big difference in a very complex **Where** clause.

Nested queries

Of all the things you can do with the **Select** statement, *nested queries* are the most complex. In a nested query, you use a second (or third or fourth) **Select** statement nested inside your main statement. Typically, nested queries are used to return a set of values that can be used with the **In** operator.

Selecting rows using the In operator

Sometimes you want to compare a column to a list of values, as shown in the query below:

```
Select *
From Customers
Where State = "ND"
   Or State = "SD"
   Or State = "MN"
```

While this is fairly easy to write, imagine the problems you might have if you had a list of 15 or 20 different values to find. An alternative to writing a bunch of different clauses is the **In** operator. The **In** operator allows you to compare a column against a set of values, as shown in the query below. It will return a list of customer names that live in North Dakota, South Dakota, or Minnesota (see Figure 4-7).

```
Select *
From Customers
Where State In ("ND", "SD", "MN")
```

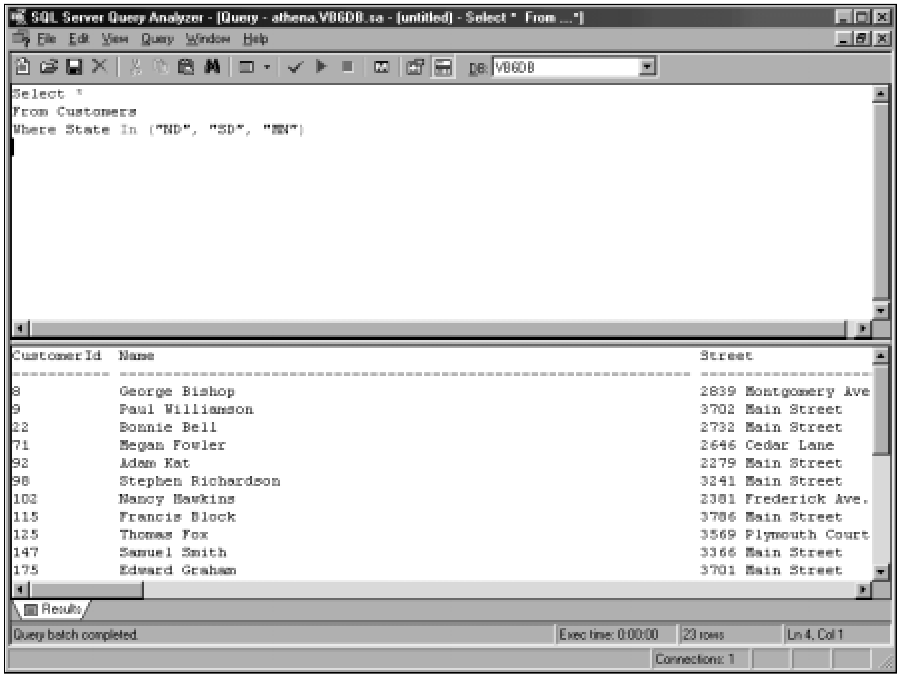
Sets of values

You can also create a set of values using a **Select** statement that can be used with the **In** operator. Consider the following query, which answers the question, “Which customers are in the same ZIP code as any of the customers that have been added since 1 January 1999?”:

```
Select Name, Zip
From Customers
Where Zip In (Select Zip
              From Customers
              Where DateAdded >= "1-January-1999")
```

While this query is somewhat contrived, it gives you an alternate way to create a set of values. You may also think that this query is similar to the one listed below, but it isn’t:

```
Select Name, Zip
From Customers
Where DateAdded >= "1-January-1999"
```


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Select * From ...*]
File Edit View Query Window Help
DB: VB6DB
Results

```
Select *
From Customers
Where State In ("ND", "SD", "MN")
```

```
CustomerId  Name                                                               Street
----------- ------------------------------------------------------------------ -------------------
8           George Bishop                                                      2839 Montgomery Ave
9           Paul Williamson                                                    3702 Main Street
22          Bonnie Bell                                                        2732 Main Street
71          Megan Fowler                                                       2646 Cedar Lane
92          Adam Kat                                                           2279 Main Street
98          Stephen Richardson                                                 3241 Main Street
102         Nancy Hawkins                                                      2381 Frederick Ave.
115         Francis Block                                                      3786 Main Street
125         Thomas Fox                                                         3569 Plymouth Court
147         Samuel Smith                                                       3366 Main Street
175         Edward Graham                                                      3701 Main Street
```

```
Query batch completed.    Exec time: 0:00:00    23 rows    Ln 4, Col 1
Connections: 1
```

Figure 4-7: Finding customers in multiple states

The two queries could only be identical if each ZIP code had only one customer.

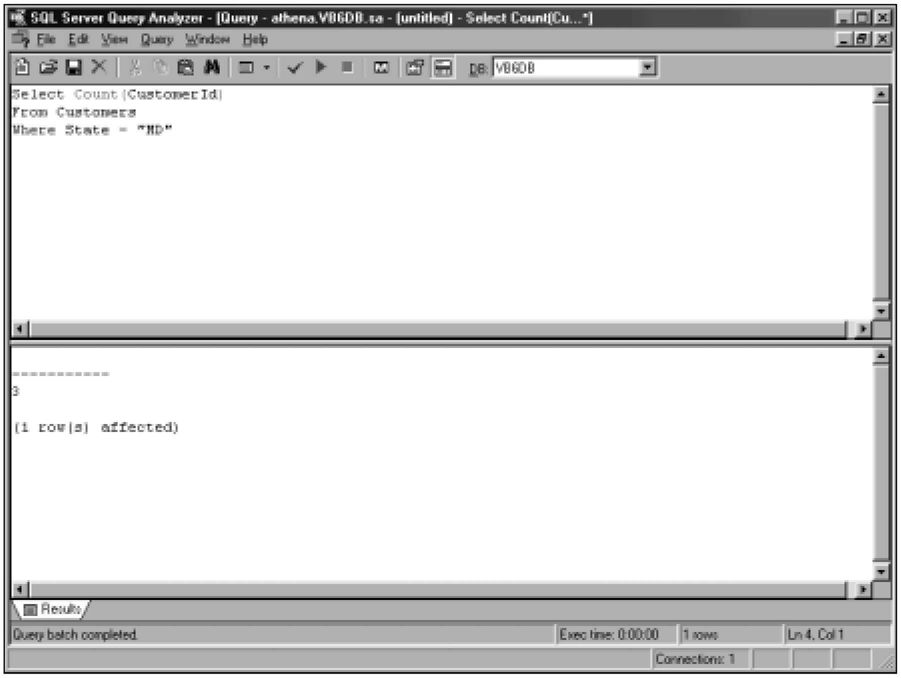
Tip

Complex to write, complex to debug: Nested queries often take a while to debug. The syntax errors will drive you nuts. I suggest that you avoid using them unless you can't do the query any other way. Unfortunately, there are some questions that you might want to ask that can only be answered using nested queries.

Using functions

You can also include various functions in your **Select** statement. For instance, the following **Select** statement counts the number of Customers who live in the state of Maryland (see Figure 4-8):

```
Select Count(CustomerId)
From Customers
Where State = "MD"
```



```
Select Count(CustomerId)
From Customers
Where State = "MD"
```

```
-----------
3

(1 row(s) affected)
```

Figure 4-8: Counting the customers from Maryland

Other functions that you can use include **Min**, **Max,** and **Average**, which will compute the minimum, maximum, or average of a particular value across all of the rows selected from the database.

Tip

Not in my program, you don't: You probably aren't going to use functions in your application program. However, using functions in an interactive query program may help you decide if your program is working. You can use the **Count** function to determine the number of rows that you just added to your table, or you can use it to count the number of rows your program updated. Sometimes, just a quick check can help you identify if you actually processed all of the rows you thought you had.

Inserting Rows into a Table

The SQL **Insert** statement is used to add one or more rows to a table. Here is the syntax for this statement:

```
Insert [Into] <table> [(<column> [, <column>} ...])]   [ Values
(<value> [,<value>]...) |
As <selectstatement> ]
```

where

```
<table> is the name of where you want to insert new rows.
<column> is the name of a column in the table.
<value> is a value that you wish to insert into a column.
<selectstatement> is a valid Select statement.
```

The **Insert** statement adds a row into the specified table. You can specify a list of columns for which you will assign the values or use the list of columns specified when the table was created. You can either explicitly specify the list of values in the **Value** clause or use the **As** keyword to specify a **Select** statement that will retrieve values from another table.

Using the **Value** clause, you will specify the list of values to be inserted into the table. The position of each value corresponds to the order of the columns specified in the **Insert** statement, or if the list columns were not specified, the values will correspond to the order of the columns in the table definition.

Using the **As** clause with a **Select** statement allows you to populate a table with data from another table. Like the **Value** clause, the columns retrieved in the **Select** statement must match up with the columns specified after the table name.

Tip

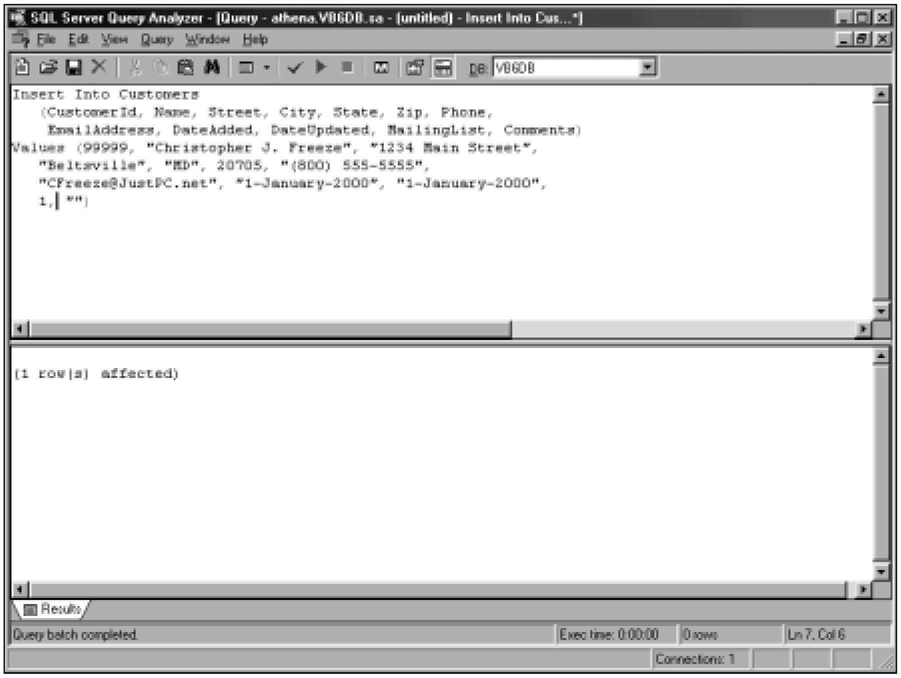
Testing with copied data: When you are testing code that deletes or updates data in a table, it is often useful to create a temporary table with a copy of your test data and use that table for your testing. This allows you to easily refresh your data after your program deletes the wrong information. Using the **Insert** statement with the **As** clause makes this very easy to do.

A simple Insert statement

Here's a very simple **Insert** statement:

```
Insert Into Customers
   (CustomerId, Name, Street, City, State, Zip, Phone,
    EmailAddress, DateAdded, DateUpdated, MailingList,
Comments)
Values (99999, "Christopher J. Freeze", "1234 Main Street",
   "Beltsville", "MD", 20705, "(800) 555-5555",
   "CFreeze@JustPC.net", "1-January-2000", "1-January-2000",
   1, "")
```

It adds a single row of information into the Customers table (see Figure 4-9). Note that I explicitly specify each of the columns in the table. Each of the values listed in the **Values** clause corresponds to the column listed in after the table name.


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Insert Into Cus...*]
File Edit View Query Window Help
DB: VB6DB

```
Insert Into Customers
   (CustomerId, Name, Street, City, State, Zip, Phone,
    EmailAddress, DateAdded, DateUpdated, MailingList, Comments)
Values (99999, "Christopher J. Freeze", "1234 Main Street",
   "Beltsville", "MD", 20705, "(800) 555-5555",
   "CFreeze@JustPC.net", "1-January-2000", "1-January-2000",
   1, "")
```

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

Results
Query batch completed.
Exec time: 0:00:00
0 rows
Ln 7, Col 6
Connections: 1

Figure 4-9: Adding a single row to the Customers table

The **Insert** statement listed below is identical to the previous one, but it assumes that the order of the columns as defined in the database is the same as the order of the data in the **Value** clause.

```
Insert Into Customers
Values (99999, "Christopher J. Freeze", "1234 Main Street",
   "Beltsville", "MD", 20705, "(800) 555-5555",
   "CFreeze@JustPC.net", "1-January-2000", "1-January-2000",
   1, "")
```

Note

To run once is good, to run twice is bad: Running this statement more than once will cause an error. Since the CustomerId field is the primary key for the table and each row must have a unique value, attempting to add another row with the same value will cause an error.

Deleting Rows from a Table

The **Delete** statement is used to remove one or more rows from a table. Here is the syntax for this statement:

```
Delete From <table>
[Where <expression>]
```

where

```
<table> is the name of the database table from which you want
to delete the rows.
<expression> is an expression that is used to determine which
rows to delete.
```

The **Delete** statement is the opposite of the **Insert** statement. It is used to remove rows from a table. The **Delete** statement uses the **Where** clause from the **Select** statement to identify which rows should be deleted.

Tip

Deleting rows: When deleting a specific row from a table, use the primary key in the **Where** clause to identify the specific row you want to delete.

A Sample Delete Statement

The following **Delete** statement will delete the row I just added (see Figure 4-10):

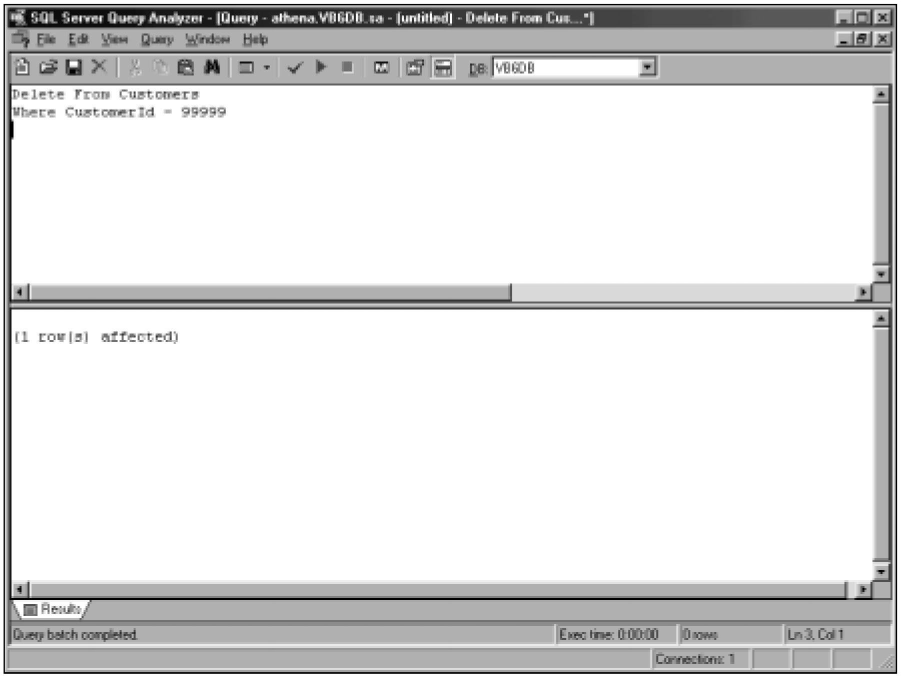
```
Delete From Customers
Where CustomerId = 99999
```

You code the **Where** clause the same way you would the **Select** statement. In this case, I only want to delete the one row, so I need to code the **Where** clause to select the specific row I want to delete.

The **Delete** statement can also be very dangerous. The following statement will delete all of the rows in the Customers table:

```
Delete From Customers
```

Note that the only difference between this statement and the previous one is the missing **Where** clause.


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Delete From Cus...*]
File Edit View Query Window Help
DB: VB6DB
Results
Query batch completed.
Exec time: 0:00:00
0 rows
Ln 3, Col 1
Connections: 1

```
Delete From Customers
Where CustomerId = 99999
```

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

Figure 4-10: Deleting a single row from the Customers table

Caution

Don't delete everything: It is very easy to delete everything from a table. For that reason, you should exercise extreme caution whenever you use the **Delete** statement. Always use the **Where** clause when using the **Delete** statement. Failure to do so will delete all of the rows in your table. Unless you are deleting data as part of a transaction (see Chapter 16, "Transactions" for more information), you can't recover any deleted records.

Updating Rows in a Table

The **Update** statement allows you to change values in one or more columns in one or more rows. Here is the syntax for this statement:

```
Update <table>
Set <column> = <value> [, <column> = <value>] ...
Where <expression>
```

where

```
<table> is the name of the table you want to update.
<column> is a column name in the table you want to update.
```

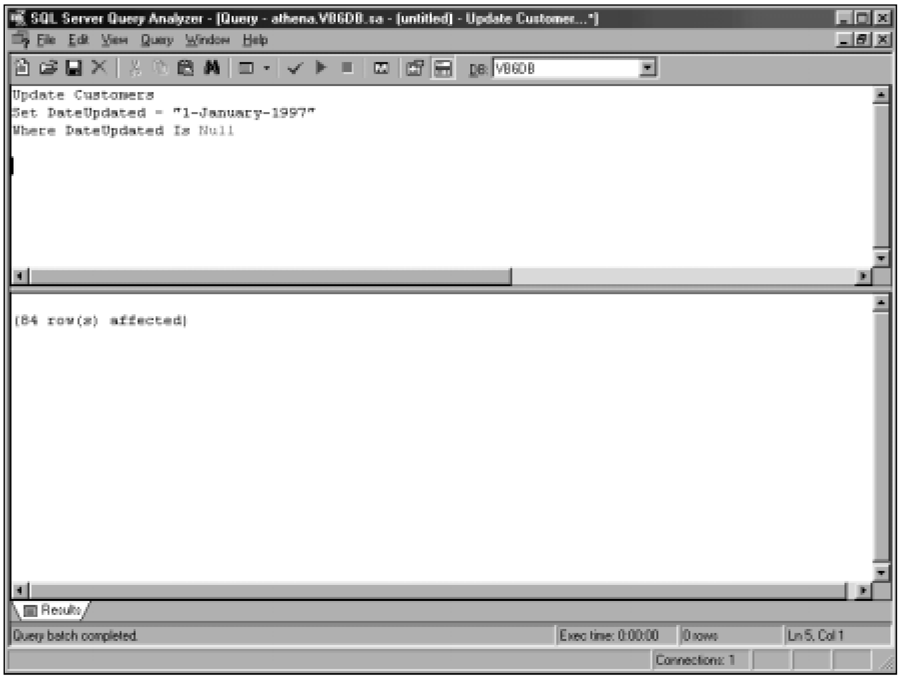
```
<value> is an expression containing the new value for the
column.
<expression> is true for the rows you want to update in the
table.
```

The **Update** statement allows you to change any value in any row in a table. Like the **Delete** statement, you need to include a **Where** clause to isolate the effects of this statement only to the rows you want to update. Otherwise, you would apply the change to all of the rows in the table.

A Sample Update Statement

The following **Update** statement will search for all rows that have a **Null** value for DateUpdated in the Customers table and replace the value with a valid date (see Figure 4-11).

```
Update Customers
Set DateUpdated = "1-January-1997"
Where DateUpdated Is Null
```


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Update Customer...*]
File Edit View Query Window Help
DB: VB6DB
Results

```
Update Customers
Set DateUpdated = "1-January-1997"
Where DateUpdated Is Null
```

```
(84 row(s) affected)
```

Query batch completed. Exec time: 0:00:00 0 rows Ln 5, Col 1 Connections: 1

Figure 4-11: Changing **Nulls** to a valid date

The Create Table Statement

The **Create Table** statement is used to build a new table in your database.

Note

There's more to this statement than meets the eye: Nearly all database vendors have added many vendor-specific extensions to the **Create Table** statement that I'll cover in more detail when I focus on the specific database systems in Chapters 23, 26, and 29.

Here is the syntax for this statement:

```
Create Table <tablename> (<columndef> [, <columndef>]...)
```

where

```
<columndef> ::= <columnname> <datatype> [Null | Not Null ]
```

and

```
<tablename> is the name of your table.
<columnname> is the name of a column in your table.
<datatype> is a valid data type.
```

The **Create Table** statement allows you to define the collection of columns that make up a table. The table must not already exist in your database, or you'll get an error message when you try to create it.

Each column must be assigned a valid data type. Table 4-2 earlier in this chapter lists some of the common data types available for a relational database.

Cross-Reference

For more detailed information about the data types available in a particular database, see Chapter 23, "Overview of SQL Server," Chapter 26, "Overview of Oracle 8*i*," or Chapter 29, "Overview of Microsoft Jet."

Tip

Gone with the table: To remove a table from your database, use the **Drop Table** <tablename> SQL statement. This statement will also delete any indexes associated with the table. Just be certain that you really want to delete the table, since it can't be undeleted.

The following SQL statement creates the Customers table for SQL Server:

```
Create Table Customers (CustomerId Int Not Null,
   Name Varchar(64), Street Varchar(64), City Varchar(64),
   State Char(2), Zip Int, Phone Varchar(32),
    EMailAddress Varchar(128), DateAdded Datetime,
    DateUpdated Datetime, MailingList Bit,
    Comments Varchar(256))
```

Note that I declare the value CustomerId as **Not Null**, since this column is the primary key for this table.

The Create Index Statement

The **Create Index** statement is used to add an index to a table in your database.

Caution

Choose carefully, my child: Picking the right set of indexes can be difficult. You should use an index on the primary key of a table, especially if you plan to retrieve rows based on the primary key. However, adding any other indexes can severely impact your database's performance, since each time you add a row to the database, the database server has to update all of the indexes. The more indexes you have, the longer it will take to update your data.

Here is the syntax for this statement:

```
Create [Unique] Index <indexname> On <tablename> (<columnname>
[, <columnname>]...)
```

where

```
<indexname> is the name of your index.
<tablename> is the name of your table.
<columnname> is the name of a column in your table.
```

The **Create Index** statement adds an index to your table using the specified columns. Using an index can improve the performance of queries that use those columns at the cost of additional work the database server must do each time any of the values in the specified columns are changed.

Including the keyword **Unique** means that the set of values included in the index will be unique in the table. This is a useful feature if you need to ensure that you don't have two or more rows with the same value in the indexed columns.

Tip

Index be gone: To remove an index from your table, use the **Drop Index** <index-name> SQL statement.

A Sample Create Index Statement

The following SQL statement creates a **Unique** index on the CustomerId field of the Customers table:

```
Create Unique Index CustomerIndex
On Customers (CustomerId)
```

This ensures that each value of CustomerId in the table will be unique and also that queries using the CustomerId column in the **Where** clause will run faster.

The Create View Statement

The **Create View** statement creates a virtual table that can be used like any other table in your database. Here is the syntax for this statement:

```
Create View <viewname> [(<columnname> [, <columnname>]...)]
As <selectstatement>
```

where

```
<viewname> is the name of your view.
<columnname> is the name of a column in your view.
<selectstatment> is valid select statement that returns the
information in your view.
```

Cross-Reference

For more information about views, refer to “Views” in Chapter 2.

The virtual table that the **Create View** statement defines in your database is indistinguishable from a regular table for any operations involving a **Select** statement. The virtual table can often be updated, depending on how the view was created.

In order to update a view, the **Select** statement must only reference a single table known as the *base table,* and it must not return any calculated values using functions and/or mathematical formulas. Also, any columns not explicitly included in the view must be able to accept **Null** values. When you try to add a row to a view, any columns in the base table that are not part of the view will be set to **Null**.

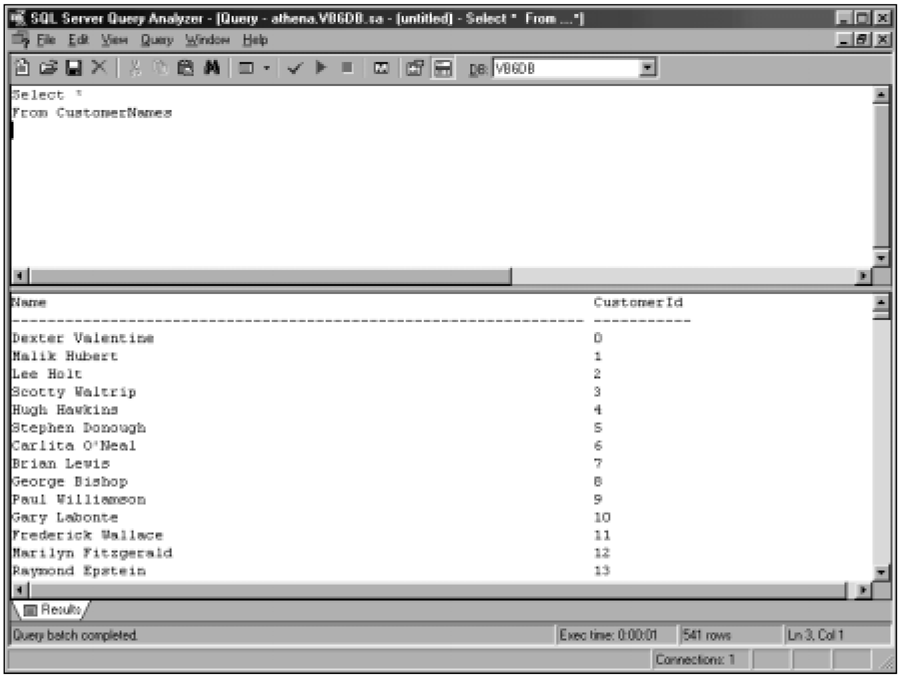
Tip

Bye-bye view: To remove a view from your table, use the **Drop View** <viewname> SQL statement.

The following SQL statement creates a view that consists of the customer’s name and the CustomerId column:

```
Create View CustomerNames As
   Select Name, CustomerId
   From Customers
```

Figure 4-12 shows the results of using a **Select** statement against the view. This view can be updated, since the columns that are not included will accept **Null** values. This technique is known as *vertical partitioning*, since only some of the columns are made available to the user.


SQL Server Query Analyzer - [Query - athena.VB6DB.sa - [untitled] - Select * From ...*]
File Edit View Query Window Help
DB: VB6DB
Results

```
Select *
From CustomerNames
```

```
Name                                                              CustomerId
----------------------------------------------------------------- -----------
Dexter Valentine                                                  0
Malik Hubert                                                      1
Lee Holt                                                          2
Scotty Waltrip                                                    3
Hugh Hawkins                                                      4
Stephen Donough                                                   5
Carlita O'Neal                                                    6
Brian Lewis                                                       7
George Bishop                                                     8
Paul Williamson                                                   9
Gary Labonte                                                      10
Frederick Wallace                                                 11
Marilyn Fitzgerald                                                12
Raymond Epstein                                                   13
```

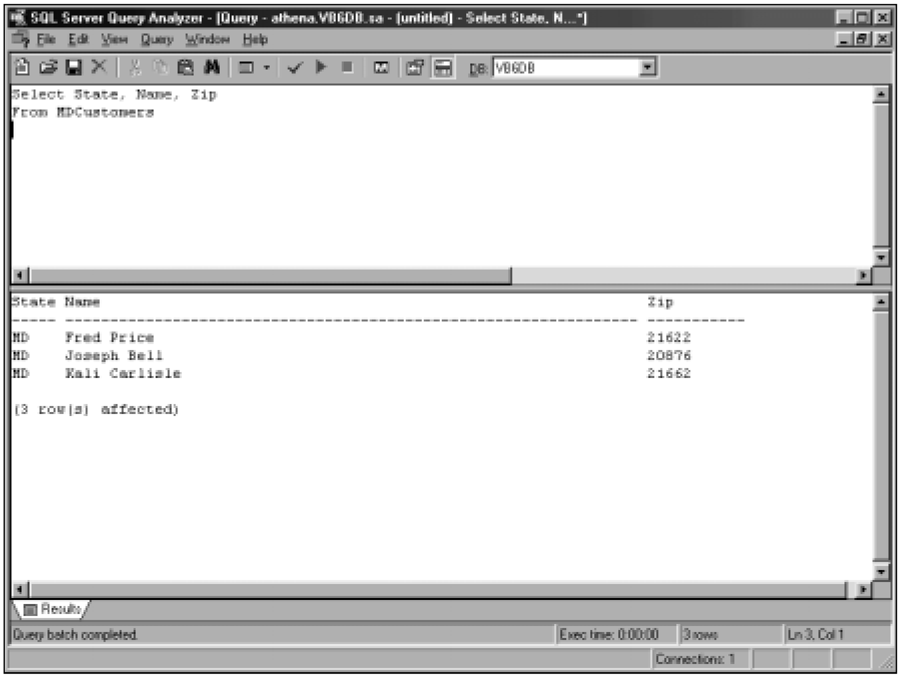
Query batch completed. | Exec time: 0:00:01 | 541 rows | Ln 3, Col 1 | Connections: 1

Figure 4-12: Preventing someone from seeing customer information beyond the customer's name and CustomerId

You can also use a **Where** clause to retrieve only some of the rows in a table. This technique is known as *horizontal partitioning*. Horizontal partitioning is useful if you need to create a view where only some of the rows in the table are retrieved. The following SQL statement creates a view containing only the customers found in Maryland (see Figure 4-13):

```
Create View MdCustomers As
   Select *
   From Customers
   Where State = "MD"
```

Note that this view is updateable, since all of the columns from the base table are present. Also, even though the view is restricted so that only customers from Maryland will be returned, you can insert rows using a different value for State. Thus, it is possible to add a row to the view that you will be unable to later retrieve.



SQL Server Query Analyzer - [Query - athena.VB6DB.sa - (untitled) - Select State, N...*]
File Edit View Query Window Help
DB: VB6DB
Results
Query batch completed.
Exec time: 0:00:00
3 rows
Ln 3, Col 1
Connections: 1

```
Select State, Name, Zip
From MDCustomers
```

```
State Name                                                                  Zip
----- --------------------------------------------------------------------- -----------
MD    Fred Price                                                            21622
MD    Joseph Bell                                                           20876
MD    Kali Carlisle                                                         21662

(3 row(s) affected)
```

Figure 4-13: Restricting the view to only the customers from Maryland

Thoughts on Using SQL to Speed Your Development Process

The **Create Table**, **Create View**, and **Create Index** statements are known as *Data Definition Language* (DDL) statements, while the **Insert**, **Delete**, **Update**, and **Select** statements are known as *Data Manipulation Language* (DML) statements. In most database systems today, you rarely execute DDL statements when you want to create a database structure. Instead, you use a utility supplied with the database system that allows you to fill in all of the information into a table or use a wizard that will help you create your table or index.

This doesn't mean that the DDL statements aren't used. It merely means that you enter the information in a different fashion. The database utility usually includes a feature that will allow you to generate the SQL statements from the definitions you entered. Then you might use these SQL statements to create a copy of the database on another computer or include them in your application if you want your users to be able to create the database structures on the fly. Visual Basic includes several object models that isolate the programmer from the underlying SQL statements. So, while you may not need to know how to use the SQL statements to write a database application, you may find yourself backed into a corner, in terms of the technology.

Simply using an interactive query tool to retrieve information from your database will help you understand if your program is working properly or not. You can use the **Select** statement to return rows from a table and you can verify that they were updated properly. When used with the **Count** function, you can find out if your program processed the correct number of rows.

The **Insert** and **Create Table** statements can be used together to create test copies of a table, with which you can test updating and deleting rows repeatedly until you are satisfied that your program is running correctly.

Most database vendors supply a rich environment for executing SQL statements. Using this environment, it is possible to write multi-statement SQL programs called *stored procedures* that perform fairly complex operations. Since these stored procedures run totally on the database server, they may run significantly faster than executing them one statement at a time from your local computer. This is an important concept to keep in mind when developing a database application, and one that I'll explore in more depth when I talk about the stored procedures.

Also, keep in mind that many OLE DB providers translate the activities you perform into SQL statements that are sent to the database server for execution. This happens even if you don't explicitly include SQL statements in your program. So, in cases where every cycle counts, you might consider coding the SQL statements yourself rather than letting the provider do the work.

Another thing to keep in mind is that this chapter contains just enough information to get you started in learning how to use SQL. Since I feel that the best way to learn is by doing, I suggest you take the time to use Query Analyzer, SQL*Plus, or any other interactive SQL utility program to practice building and executing SQL statements. While I believe in the KISS rule (Keep it Simple Stupid!), there are times where it is appropriate to use very complex SQL statements. And the best way to write complex SQL statements that work is by practicing writing simple SQL statements.

Summary

In this chapter you learned:

- ✦ about the SQL statements and data types.
- ✦ how to use the **Select** statement to retrieve information from your database.
- ✦ how to use the **Insert**, **Delete,** and **Update** statements to manipulate the data in your database.
- ✦ how to use the **Create Table** statement to create a new table.

- how to use the **Create Index** to allow the database to find specific rows in your table more quickly.
- how to use the **Create View** statement to create a virtual table that can be used just like a real table, but whose contents are dynamically created from other tables in the database.

✥ ✥ ✥