10. STRUCTURED QUERY LANGUAGE

Structured Query Language (SQL) is a computer language for communication and interaction with databases and was created to be a cross-platform syntax to extract and manipulate data from disparate database systems. So in theory the same SQL queries written for an Oracle database will work on a Sybase database or an Access database and so on. However, database vendors have also developed their own versions of SQL such as Transact-SQL and Oracle's PL/SQL which extend ANSI/ISO SQL. This chapter will focus on writing standard SQL and will not use any vendor specific SQL syntax.

In our database connection program in the previous chapter, the communicating parties were a "front end," our VC++.NET application that sent an SQL statement via an OleDbDataAdapter, and a "back end" data source that held the data. The statement, the SQL code, can contain instructions that can read or change the data within the database or manipulate the database itself in some way. We can embed SQL statements into our VC++.NET programs to perform everything from simple data retrieval to high-level operations on databases.

The SQL statement that we may most often be concerned with when developing quantitative trading or risk management systems are those that retrieve data, called queries. However, we will at times also need to write, change or delete data in a database. By the end of this chapter, you should have a good understanding of the basic syntax of SQL, which consists of only several dozen or so words. SQL statements are simply groups of those words logically arranged to pull specific data from the data source or manipulate the data in the data source. These types of SQL statements are referred to as data manipulation language (DML). Also, however, SQL can be used to actually manipulate the database itself. These SQL statement are called data definition language (DDL).

10.1 DATA MANIPULATION LANGUAGE

We use DML to retrieve, alter and otherwise work with the actual data held within a database.

10.1.1 THE SELECT STATEMENT

Reading data is the most common task we want to perform against a database. A SELECT statement queries the database and retrieves selected data that matches the criteria that we specify. The SELECT statement has five main clauses, although a FROM clause is the only required one. Each of the clauses has a wide array of options and parameters. Here, we will show the general structure of a SELECT statement with clauses. However, each of them will be covered in more detail later in the chapter.

```
SELECT [ALL | DISTINCT] column1,column2
FROM table1,table2
[WHERE "conditions"]
[GROUP BY "column-list"]
[HAVING "conditions”]
[ORDER BY "column-list" [ASC (Default) | DESC] ];
```

A SELECT statement means that we want to choose columns from a table. When selecting multiple columns, a comma must delimit each of them except for the last column. Also, be aware that as with VC++.NET, SQL is not case sensitive. Upper or lower case letters will do just fine. Be aware too that most, but not all, databases require the SQL statement to be terminated by a semi-colon.

Before we get too in depth, let's create a VC++.NET program to test out the SQL statements we look at as we go along.

STEP 1 Create a new VC++ Windows Application named SQL_Example.

STEP 2 To your Form1, add a textbox, three buttons and a large DataGridView.

STEP 3 In the Form1 code window add the following code. Notice that we will be using the Options.mdb database.

```
Imports System.Data.OleDb

Public Class Form1
    Private m_Connection As OleDbConnection
    Private m_Adapter As OleDbDataAdapter
    Private m_DataSet As DataSet


    Private Sub Button1_Click( … ) Handles Button1.Click
        m_Connection = New OleDbConnection( _
                   "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Temp\Finance.mdb")
        Try
            m_Connection.Open()
            m_Adapter = New OleDbDataAdapter(TextBox1.Text, m_Connection)
            m_DataSet = New DataSet
            m_Adapter.Fill(m_DataSet)
            DataGridView1.DataSource = m_DataSet.Tables(0)
        Catch ex As Exception
            MessageBox.Show("Enter a valid SQL statement.")
        Finally
            m_Connection.Close()
        End Try

    End Sub
End Class
```

This program will allow the you to provide an SQL statement during runtime. Furthermore, we will be able to test out several SQL statements without having to rerun the program as we have also included a Try…Catch block so the program won't crash if you make a mistake in the SQL statement.

STEP 4 Run the program and enter the simple SQL statement shown below in the textbox.

```
SELECT OptionSymbol,StockSymbol,Year,Month,Strike,Bid,Ask,OpenInt
FROM OptionContracts;
```

STEP 5 Click button1. For each of the following SQL statements, we will always be clicking button1.

Note that the columns are displayed in the order that they appear in the SELECT statement. If all columns from a table were needed to be part of the result set, we do not need to explicitly specify them. Rather, in the case where all columns are to be selected, we can use the * symbol. As we saw in the previous chapter's example, the resulting SQL statement would look like this:

```
SELECT * FROM OptionContracts;
```

For now, leave your SQL_Example Windows application running. You can test out the SQL statements as you read through the rest of the chapter.

10.1.2 THE WHERE CLAUSE

The previous example retrieved a result set that included all the rows in the table from the specified columns. However, we may want to filter some rows out according to some condition or based upon some comparison. This is where the WHERE clause comes in. For comparison in SQL, we use the following operators:

Comparison Operator	Description
<	Contents of the field are less than the value.
<=	Contents of the field are less than or equal to the value.
>	Contents of the field are greater than the value.
>=	Contents of the field are greater than or equal to the value.
=	Contents of the field are equal to the value.
<>	Contents of the field are not equal to the value.
BETWEEN	Contents of the field fall between a range of values.
LIKE	Contents of the field match a certain pattern.
IN	Contents of the field match one of a number of criteria.

So, if we were interested in only the option contracts with open interest greater than 1,000, our SQL would look like this:

```
SELECT * FROM OptionContracts WHERE OpenInt > 1000;
```

The WHERE clause can also have multiple conditions using AND or OR. If we wanted to see all contracts where open interest is over 1,000 and the bid is greater than 0, it would look like this:

```
SELECT * FROM OptionContracts WHERE OpenInt > 1000 AND Bid > 0;
```

If we needed to build a WHERE clause for such a field, MS Access requires that we use single quotes for string comparison like this:

```
SELECT * FROM OptionContracts WHERE StockSymbol = 'IBM';
```

Date comparison requires the use of the pound sign, #. For example, if we wanted to see all of the options trades done in February of 2003:

```
SELECT * FROM OptionTrades
WHERE TradeDateTime >= #2/01/2003# AND TradeDateTime <= #2/28/2003#;
```

10.1.3 The ORDER BY Clause

We can sort our result set with the ORDER BY clause. ORDER BY is an optional clause, which allows us to display the results of our query in a sorted order, either ascending or descending, based on the columns that we specify to order by. Here is an example:

```
SELECT * FROM OptionContracts
```

```
WHERE StockSymbol = 'MSFT' ORDER BY OpenInt;
```

This statement selects all the MSFT option contracts and orders the data by from the lowest open interest to the highest. To view the data in descending order, we simply add DESC to the end.

```
SELECT * FROM OptionContracts
WHERE StockSymbol = 'MSFT' ORDER BY OpenInt DESC;
```

If we need to order based on multiple columns, we must separate the columns with commas.

```
SELECT * FROM OptionContracts
WHERE StockSymbol = 'MSFT' ORDER BY OpenInt DESC, Strike;
```

Notice now that the contracts that have the same open interest are now listed in order of strike price. Also, the DESC applies only to the OpenInt. Strike is sorted with the default ASC order.

10.1.4 THE LIKE CLAUSE

So far we have learned how to find exact matches with SQL. However, there may be times that we need to search for partial strings. SQL provides a LIKE operator for just this type of query.

The LIKE operator can only be used on fields that have one of the string types set as their data type. LIKE cannot be used on dates or numbers.

String comparison employs a wildcard sign, %, which can be used to match any possible character that might appear before or after the characters specified. If you want to view all of the IBM option contracts with an 80 strike:

```
SELECT * FROM OptionContracts
WHERE StockSymbol = 'IBM' AND OptionSymbol LIKE '%P';
```

The LIKE operator proves to be very useful as we write more complex SQL statements, since it enables us to find partial matches without performing any complicated string manipulation. Keep in mind, however, that the LIKE operator is not the most efficient SQL command, and will degrade overall performance. If we know the exact string that we are looking for in a field, then we should use the = operator instead of LIKE.

In addition to the % wildcard, there are two other important wildcards used with the LIKE operator--the underscore (_) and the square brackets ([]). Whereas the % wildcard is used to find a string with any number of characters before and/or after the specified characters, the underscore is used to limit the search to a single leading or trailing character. A search of '%D%' would return MCDRE and IBMDP. Say, for example, we wanted to find all the April calls for all the stocks. Omitting the IBM WHERE class and changing the LIKE expression to '%D_' would limit the return values to just those calls with April expiration, since we are now looking for any option contract with a D as the second to last letter in its symbol.

```
SELECT * FROM OptionContracts WHERE OptionSymbol LIKE '%D_';
```

Additionally, we can use the brackets ([]) to further limit ranges of characters. With the brackets, we can specify particular characters that must appear in a particular position. For instance, if we were looking for April and May calls, then we need to modify our criteria. We limit our search to option contracts that have either D or E in the second to last position, so we specify this by putting these characters within brackets, in the appropriate place:

```
SELECT * FROM OptionContracts WHERE OptionSymbol LIKE '%[DE]_';
```

Keep in mind that the brackets may only contain single characters, so we cannot use them for lists of strings. This is the biggest limitation to the bracket wildcard, but there are still a large number of possibilities for expression searching in strings.

10.1.5 AGGREGATE SQL FUNCTIONS

So far the SQL that we have been using retrieves rows of data from the database. But, SQL can do a lot more. Among other things, SQL has a few built in functions that can tell us things about the data as a whole. For example, what if we wanted to know what contract has the largest open interest? How about the total number of trades for a given month? As you can see, these numbers are not contained within the columns of a table. Rather, they must be computed.

ANSI/ISO SQL contains aggregate functions that can compute simple information from the data in a database. Specific RDBMSs may support additional aggregate functions that are proprietary and also very useful. Supported ANSI/ISO SQL aggregate functions.

Aggregate Function	Description
AVG	Returns the average of the values in a column.
COUNT	Returns the total number of values in a column.
COUNT(*)	Returns the number of rows in a table.
MAX	Returns the largest value in a column.
MIN	Returns the smallest value in a column.
SUM	Returns the sum of the numeric values in a column.

10.1.5.1 THE SUM FUNCTION

Let's begin by taking a look at the SUM function. It is used within a SELECT statement and, predictably, returns the summation of a series of values. In this example, we will compute the total number of shares traded in the month of January, 2003.

```
SELECT SUM(Quantity) FROM StockTrades
WHERE TradeDateTime >= #1/1/2003# AND TradeDateTime <= #1/31/2003#;
```

```
SELECT * FROM StockTrades
WHERE TradeDateTime BETWEEN #01/01/2003# AND #1/31/2003#;
```

Notice that the result set only contains one row of data. This is to be expected when using any of the SQL aggregate functions. Also notice the name of the column. Since we asked SQL to return an aggregate value, SQL named the column for us. When this occurs, we say that an SQL computed column is being used.

Of course, the column name ‘Expr1000’ is not descriptive of the data it contains. Fortunately, SQL column naming is simple. To rename computed columns, use the AS modifier. The AS modifier allows us to give meaningful names to any computed columns. If we wanted to give a meaningful name, say TotalShares, to the computed column in the SQL statement written above, we could write it as:

```
SELECT SUM(Quantity) AS TotalShares FROM StockTrades
WHERE TradeDateTime >= #1/1/2003# AND TradeDateTime <= #1/31/2003#;
```

10.1.5.2 THE AVG / COUNT / MIN / MAX FUNCTIONS

Predictably, these aggregate functions will return the average of the data in a column, the lowest and highest values in a column, and the count or number of elements in a column. If we wanted to obtain the respective values for the month of January, 2003, our SQL statements would look as follows:

```
SELECT MIN(Quantity) FROM StockTrades
WHERE TradeDateTime >= #01/01/2003# AND TradeDateTime <= #1/31/2003#;
```

```
SELECT MAX(Quantity) FROM StockTrades
WHERE TradeDateTime >= #01/01/2003# AND TradeDateTime <= #1/31/2003#;
```

```
SELECT AVG(Quantity) FROM StockTrades
WHERE TradeDateTime >= #01/01/2003# AND TradeDateTime <= #1/31/2003#;
```

```
SELECT COUNT(*) FROM StockTrades
WHERE TradeDateTime >= #01/01/2003# AND TradeDateTime <= #1/31/2003#;
```

10.1.6 THE DISTINCT FUNCTION

The SQL DISTINCT function is useful when only the first occurrence of a desired series of data is needed. For example, if we were interested in seeing a list of all the stocks that have been traded, we would not care to see duplicates. That is, we may have traded MSFT several times and we don't care to see it listed more than once. We can filter out duplicates with the DISTINCT function.

```
SELECT DISTINCT(StockSymbol)
FROM StockTrades
ORDER BY StockSymbol;
```

10.1.7 THE GROUP BY CLAUSE

As we have just seen, using aggregate functions such as SUM and MIN will get us a computed value for a group of records. What if, however, we wanted to write a SQL statement that would show the SUMs of the quantities traded each individual option symbol? The GROUP BY will return the results of aggregate functions for a group of values.

```
SELECT OptionSymbol,SUM(Quantity)
FROM OptionTrades
GROUP BY OptionSymbol;
```

Notice that option symbols are only displayed when they have a greater than zero value. If for example, the summation of the quantity for AXPDZ were zero, it would not be included in the result set. The GROUP BY clause can only be used when selecting multiple columns from a table or tables and at least one aggregate function appears in the SELECT statement.

When there are multiple columns beyond the one being aggregated, we can GROUP BY all the other selected columns. For example, if we wanted the total quantity for all option symbols by BuySell, the SQL would look like the following:

```
SELECT OptionSymbol,BuySell,SUM(Quantity)
FROM OptionTrades
GROUP BY OptionSymbol,BuySell;
```

Note that the above SQL has two columns in the GROUP BY clause. Remember, if the column appears in the SELECT and the SELECT has aggregate functions, the column must appear in a GROUP BY clause.

10.1.8 The HAVING Clause

The HAVING clause is like a WHERE clause for groups. By definition, an SQL statement that uses a GROUP BY clause cannot use a WHERE clause. We must use a HAVING clause instead. For example, if we wanted to see only those option contracts that have total quantities traded of greater than or equal to 50, the SQL statement would look like this.

```
SELECT OptionSymbol,SUM(Quantity)
FROM OptionTrades
GROUP BY OptionSymbol HAVING SUM(Quantity) >= 50;
```

The HAVING clause is reserved for aggregate functions and is usually placed at the end of an SQL statement. Also, an SQL statement with a HAVING clause may or may not necessarily include the GROUP BY clause. The following SQL statement is valid:

```
SELECT COUNT(OptionSymbol)
FROM OptionTrades
HAVING SUM(Quantity) >= 50;
```

10.1.9 Mathematical Operations

We can also make mathematical calculations in our SQL statements—add, subtract, multiply, divide, and modulus. The following SQL uses a column alias to describe the (quantity * price) of a trade:

```
SELECT OptionSymbol,Price,Quantity,(Price * Quantity * 100)
AS TradeCost
FROM OptionTrades;
```

10.1.10 Aliasing

Tables in a database can be aliased in a FROM clause. The following example creates an alias named “OT” for the OptionTrades table.

```
SELECT * FROM OptionTrades OT;
```

This is convenient when you want to retrieve information from two or more separate tables (an operation known as joining). The advantage of using a table alias when joining will become apparent as our SQL statements become more complex.

10.1.11 Joining Tables

So far in our examples, we have retrieved data from only one table. In many instances, however, we may need to retrieve data from two or more tables.

The Stock table and the OptionContracts table above contain information about individual stocks and options contracts on those stocks and we may be interested in returning data from both tables in a single SQL statement. To join these two tables we must first identify a column in each table that contains the same data. In this example, the OptionContracts table contains a StockSymbol column, which matches the StockSymbol column in the Stock table.

The two tables can be joined on these StockSymbol columns, although it is just a coincidence that both these tables are named the same. In order to join tables, the data must match, not necessarily the column names. Here is an example below using table aliasing for readability.

```
SELECT * FROM Stock S, OptionContracts OC
WHERE S.StockSymbol = OC.StockSymbol;
```

In this example the join is performed within the WHERE clause. The above SQL will return all columns for each table joined by the stock symbol. With the two tables joined, the SELECT and the WHERE clause can now be modified. For example:

```
SELECT OC.OptionSymbol,OC.StockSymbol,OC.Bid,OC.Ask,S.DividendAmount
FROM Stock S, OptionContracts OC
WHERE S.StockSymbol = OC.StockSymbol AND S.StockSymbol = 'IBM';
```

10.1.11.1 The UNION Keyword

A UNION is useful if you want to get data from two tables within the same result set. For example, if we wanted to see the bid and ask for INTC as well as the bids and asks for all the INTC options in one result set, the SQL statement would read as follows:

```
SELECT StockSymbol,Bid,Ask FROM Stock
WHERE StockSymbol = 'IBM'
UNION
SELECT OptionSymbol,Bid,Ask FROM OptionContracts
WHERE StockSymbol = 'IBM';
```

The data type for the columns in each SELECT statement must match for a UNION to work. This is not an issue in the above example because each of the tables have identical column sets.

10.1.11.2 The Inner Join Clause

If we need to include data from two or more tables in the result set, we can use the INNER JOIN or OUTER JOIN statements SQL to do so. An INNER JOIN allows us to join two tables that have values from one or more columns in common. An OUTER JOIN allows us to join two tables that have one or more columns in common. That is, the difference between an INNER JOIN and an OUTER JOIN is that, in an INNER JOIN records from the first, or source, table that do not have a match in the second, or joining, table are excluded from the result set. In an OUTER JOIN, records from the source table that do not have a match in the joining table are still included in the result set.

We use an INNER JOIN to join two tables that have values in common from at least one column.

```
SELECT ST.StockSymbol,ST.Price,OC.OptionSymbol,OC.Bid,OC.Ask
FROM StockTrades ST
INNER JOIN OptionContracts OC
ON ST.StockSymbol = OC.StockSymbol
```

In the statement above, the StockTrades and OptionContracts tables are joined based on values in their common StockSymbol column. For rows in the StockTrades table that have a match in the OptionContract table, the data for the matching rows is included. Rows in the StockTrades table that do not have a match in the OptionContract table are excluded by the Inner Join.

10.1.11.3 The Outer Join Clause

We use an OUTER JOIN to join two tables that have in common at least one column. When the fields from the two tables are compared in the ON clause, those fields in the joining table for records in the source table that do not match will have null values.

Here is an example of an OUTER JOIN:

```
SELECT S.StockSymbol,S.Bid,S.Ask,ST.Price
FROM Stock S
LEFT OUTER JOIN StockTrades ST
ON S.StockSymbol = ST.StockSymbol
```

Notice that for rows from the Stock table that do not have a matching value between the StockSymbol column and the StockSymbol column from StockTrades, the StockTrades.Price column contains a null value. The Price field for KO is null since no trades were placed in that stock.

The use of the LEFT modifier in the OUTER JOIN means that all rows from the table on the left of the OUTER JOIN operator, that is, the Stock table, will be included in the result set, whether or not there are matches in the table to the right, the StockTrades table. If, on the other hand, we used the RIGHT modifier, all rows from the table on the right of the OUTER JOIN will be included with or without matches. Finally, we could also use the FULL modifier so that all rows from the both tables will be included in the result set.

10.1.12 The INSERT Statement

Up to this point, we have only queried the Option.mdb database and looked at the results. We may, however, also be interested in changing the data. In order to add, delete or modify the data in the Options.mdb database we will first need to add some elements to our SQL_Example program.

STEP 6: Add another button to your form.
STEP 7: Add the following code to the button2_Click event.

```
Private Sub button2_Click(…) Handles Button2.Click
```

```
 Try

   m_Connection.Open();
   Dim m_Command as OleDbCommand = new OleDbCommand( textBox1.Text, m_Connection )
   m_Command.ExecuteNonQuery()

 Catch ex as Exception
      MessageBox.Show( "Enter a valid SQL statement." )
 Finally
      m_Connection.Close()
End Sub
```

An OleDbCommand object is an SQL statement, which we can use to perform transactions against a database. We use the ExecuteNonQuery() member method to execute UPDATE, INSERT, and DELETE statements.

For the remainder of the chapter, SELECT statements should be executed using the first button, and all other transactions should be executed using this new, second button.

The SQL INSERT statement enables us to add data to a table in a database. Here is an example showing the syntax for adding a record to the OptionTrades table:

```
INSERT INTO OptionTrades
(TradeDateTime, OptionSymbol, BuySell, Price, Quantity, TradeStatus)
VALUES (#02/27/2003#,'IBMDP','B',2.60,10,'F');
```

STEP 8: Click button2. For the remainder of the chapter, we will be clicking button2. You can verify that this data has been added to the table by writing a simple SELECT statement on the OptionTrades table and clicking button1.

Notice that all values for all columns have been supplied save for the TradeID column, which is generated automatically. If a value for a column is to be left blank, the keyword NULL could be used to represent a blank columns value.

In regards to data types notice that strings are delimited by single quotes, numerical data does not need single quotes, and dates are defined with pound signs. As we have mentioned previously, each RDBMS is different, and so you should look into the documentation of your system for how to define the data types. Whatever your RDBMS, the comma-delimited list of values must match the table structure exactly in the number of attributes and the data type of each attribute.

10.1.10 The UPDATE Statement

The SQL UPDATE clause is used to modify data in a database table existing in one or several rows. The following SQL updates one row in the stock table, the dividend amount for IBM:

```
UPDATE Stock SET DividendAmount = .55
WHERE StockSymbol = 'IBM';
```

SQL does not limit us to updating only one column. The following SQL statement updates both the dividend amount and the dividend date columns in the stock table.

```
UPDATE Stock SET DividendAmount = .50,DividendDate = #03/18/2003#
WHERE StockSymbol = 'IBM';
```

The update expression can be a constant, any computed value, or even the result of a SELECT statement that returns a single row and a single column. If the WHERE clause is omitted, then the specified attribute is set to the same value in every row of the table. Also, we can also set multiple attribute values at the same time with a comma-delimited list of attribute-equals-expression pairs.

10.1.14 The DELETE Statement

As its name implies, we use an SQL DELETE statement to remove data from a table in a database. Like the UPDATE statement, either single rows or multiple rows can be deleted. The following SQL statement deletes one row of data from the StockTrades table:

```
DELETE FROM StockTrades
WHERE TradeID = 40;
```

The following SQL statement will delete all records from the StockTrades table that represent trades before January 4, 2003.

```
DELETE FROM StockTrades
WHERE TradeDateTime < #01/04/2003#;
```

If the WHERE clause is omitted, then every row of the table is deleted, which of course should be done with great caution.

We can avoid writing INSERT, UPDATE and DELETE statements by using the functionalities of a DataAdapter object as we have seen.

10.2 Updating a Database with Changes in a DataSet

As we saw in the previous chapter on ADO.NET, we can also make changes to the data in the database. This method uses the OleDbDataAdapter's Update function to automatically create and execute the proper SQL INSERT, UPDATE, or DELETE statements for each inserted, updated, or deleted row in the DataSet.

```
m_Connection.Open()
Dim m_Builder as New OleDbCommandBuilder( m_Adapter );

// Change some data in the DataSet
m_DataSet.Tables( 0 ).Rows( 0 )( 5 ) = 200;

// Call the Update method.
int m_Int = m_Adapter.Update( m_DataSet.Tables( 0 ) )
m_Connection.Close()
```

10.3 Data Definition Language

We use DDL to create or modify the structure of tables in a database. When we execute a DDL statement, it takes effect immediately. Again, for all transactions, you should click Button2 to execute these non-queries. You will be able to verify the results of the SQL statements by creating simple SELECT statements and executing a query with Button1 in your program.

10.3.1 Creating Views

A view is a saved, read-only SQL statement. Views are very useful when you find yourself writing the same SQL statement over and over again. Here is a sample SELECT statement to find all the IBM option contracts with an 80 strike:

```
SELECT * FROM OptionContracts
WHERE StockSymbol = 'IBM' AND OptionSymbol LIKE '%P';
```

Although not overly complicated, the above SQL statement is not overly simplistic either. Rather then typing it over and over again, we can create a VIEW. The syntax for creating a view is as follows:

```
CREATE VIEW IBM80s AS SELECT * FROM OptionContracts
WHERE StockSymbol = 'IBM' AND OptionSymbol LIKE '%P';
```

The above code creates a VIEW named IBM80s. Now to run it, simply type in the following SQL statement.

```
SELECT * FROM IBM80s;
```

Views can be deleted as well using the DROP keyword.

```
DROP VIEW IBM80s;
```

10.3.2 Creating Tables

As you know by now database tables are the basic structure in which data is stored. In the examples we have used so far, the tables have been pre-existing. Often times, however, we need to build a table ourselves. While we are certainly able to hild tables ourselves with an RDBMS such as MS Access, we will cover the SQL code to create tables in VC++.NET.

As a review, tables contain rows and columns. Each row represents one piece of data, called a record, and each column, called a field, represents a component of that data. When we create a table, we need to specify the columns names as well as their data types. Data types are usually database specific but usually can be broken into the following: integers, numerical values, strings, and Date/Time. The following SQL statement builds a simple table named Trades:

```
CREATE TABLE MyTrades
(MyInstr Char(4) NOT NULL,
 MyPrice Numeric(8,2) NOT NULL,
 MyTime Date NOT NULL);
```

The general syntax for the CREATE TABLE statement is as follows:

```
CREATE TABLE TableName (Column1 DataType1 Null/Not Null, …);
```

The data types that you will use most frequently are the VARCHAR2(n), a variable-length character field where n is its maximum width; CHAR(n), a fixed-length character field of width n; NUMERIC(w.d), where w is the total width of the field and d is the number of places after the decimal point (omitting it produces an integer); and DATE, which stores both date and time in a unique internal format. NULL and NOT NULL indicate whether a specific field may be left blank.

Tables can be dropped as well. When a table is dropped, all of the data it contains is lost.

```
DROP TABLE MyTrades;
```

10.3.3 Altering Tables

We have already seen that the INSERT statement can be used to add rows. Columns as well can be added or removed from a table. For example, if we wanted to add a column named Exchange to the StockTrades table, we can use the ALTER TABLE statement. The syntax is supplied below:

```
ALTER TABLE StockTrades ADD Exchange char(4);
```

As we have seen in the previous chapter, all tables must have a primary key. We can use the ALTER TABLE statement to specify TradeID in the Trades table we created previously.

```
ALTER TABLE Trades ADD PRIMARY KEY(TradeID);
```

Columns can be removed as well using the ALTER TABLE statement.

```
ALTER TABLE StockTrades DROP Exchange;
```

10.4 Summary

Over the course of this chapter, we have taken a pretty good look at SQL, from simple select statements to joining tables to updating databases and finally altering the structure of a database. Because financial markets produce so much data every day, it is important to understand SQL as a tool to interacting with large relational databases for research and testing.

Chapter Problems

1.) What is SQL? What are DDL and DML?
2.) What document should you consult to find out the specifics of SQL transactions against your RDBMS?
3.) What is an OleDbCommand object and what is the ExecuteNonQuery() method?
4.) If we found corrupt data in a database, what statements might we use to either correct it or get rid of it?
5.) What is the syntax of CREATE TABLE?

Project One

The Finance.mdb database contains price data. However, we very often will be interested in a time series of log returns. Create a VB.NET application that will modify the AXP table to include a Returns column. Then, make the calculations for the log returns and populate the column.

Project Two

Create a VB.NET application that will connect to the Finance.mdb database and return the average volume for a user-defined stock between any two user-defined dates.