BONUS CHAPTER 5

SQL Primer

In several of the chapters on working with databases, you saw how SQL statements were used to determine what information would be available in a recordset. This chapter explains how to create those SQL statements and how to do much more with SQL. The examples in this chapter all use an Access database, but the techniques of using SQL are applicable to many database formats. In fact, SQL statements are the cornerstone of working with many database servers, such as Oracle or SQL Server.

Two basic types of SQL statements are covered in this chapter: data-manipulation language (DML) and data-definition language (DDL). Most of the chapter deals with DML statements, and, unless a statement is identified otherwise, you should assume that it is a DML statement. ■

What is SQL?

SQL allows you to quickly retrieve or modify groups of records in your database.

Retrieve selected records

By setting the appropriate clauses, you can work with only a portion of a table rather than have to work with the entire table.

Get information from multiple tables

Using SQL statements allows you to easily combine information from two or more tables.

Calculate summary information

You can find out how many records are in a recordset, or determine the total or average values of specific fields.

Use SQL to modify the information in tables

With a single SQL statement, you can change the values of multiple records in a database. To do the same thing with a program would require a number of statements.

Use SQL to change the structure of the database

SQL statements can even be set up to create a table, modify the structure of a table, or delete a table.

Defining SQL

Structured Query Language (SQL) is a specialized set of programming commands that enable the developer (or end user) to do the following kinds of tasks:

- Retrieve data from one or more tables in one or more databases
- Manipulate data in tables by inserting, deleting, or updating records
- Obtain summary information about the data in tables, such as totals; record counts; and minimum, maximum, and average values
- Create, modify, or delete tables in a database (Access databases only)
- Create or delete indexes for a table (Access databases only)

SQL statements enable the developer to perform functions in one line or a few lines of code that would take 50 or 100 lines of standard BASIC code to perform.

What SQL Does

As the name implies, Structured Query Language statements create a query that is processed by the database engine. The query defines the fields to be processed, the tables containing the fields, the range of records to be included, and, for record retrieval, the order in which the returned records are to be presented.

When retrieving records, a SQL statement usually returns the requested records in a *dynaset*. Recall that a dynaset is an updatable recordset that actually contains a collection of pointers to the base data. Dynasets are temporary and are no longer accessible after they are closed. SQL does have a provision for the times when permanent storage of retrieved records is required.

NOTE The Microsoft SQL syntax used in this chapter is designed to work with the Jet database engine and is compatible with ANSI SQL (there are, however, some minor differences between Microsoft SQL and ANSI SQL). In addition, if you use SQL commands to query an external database server such as SQL Server or Oracle, read the documentation that comes with the server to verify that the SQL features you want to use are supported and that the syntax of the statements is the same. ■

The Parts of the SQL Statement

A SQL statement consists of three parts:

- *Parameter declarations* These optional parameters are passed to the SQL statement by the program.
- *The manipulative statement* This part of the statement tells the Query engine what kind of action to take, such as `SELECT` or `DELETE`.
- *Options declarations* These declarations tell the Query engine about any filter conditions, data groupings, or sorts that apply to the data being processed. These include the `WHERE`, `GROUP BY`, and `ORDER BY` clauses.

These parts are arranged as follows:

```
[Parameter declarations] Manipulative statement [options]
```

The parameter declarations section is where you define any parameters used in the SQL statement. Any values defined in the parameter declarations section are assigned before the SQL statement is executed. See the section, "Using Parameters," later in this chapter for a more detailed discussion of the parameters declaration.

Most of this chapter uses only the manipulative statement and the options declarations. By using these two parts of the SQL statement, you can create queries to perform a wide variety of tasks. Table B5.1 lists four of the manipulative clauses and their purposes.

Table B5.1 Parts of the Manipulative Statement

Statement	Function
DELETE FROM	Removes records from a table
INSERT INTO	Adds a group of records to a table
SELECT	Retrieves a group of records and places the records in a dynaset or table
UPDATE	Sets the values of fields in a table

Although manipulative statements tell the database engine what to do, the options declarations tell it what fields and records to process. The discussion of the optional parameters makes up the bulk of this chapter. In this chapter, you first look at how the parameters are used with the SELECT statement and then apply the parameters to the other manipulative statements. Many examples in this chapter are based on the sales-transaction table of a sample database that might be used to manage an aquarium business.

The following discussions of the different SQL statements show just the SQL statement syntax. Be aware that these statements can't be used alone in Visual Basic. The SQL statement is always used to create a QueryDef, to create a dynaset or snapshot by using the Execute method, or as the RecordSource property of a data control. This section explains the part of a SQL statement. Later in the chapter, the "Using SQL" section explains how these statements are actually used in code. For other examples of using SQL statements, look back through Chapters 29, "Using the Visual Basic Data Control," 30, "Doing More with Bound Controls," and 31, "Improving Data Access with Data Access Objects (DAO)."

NOTE A QueryDef is a part of the database that stores the query definition. This definition is the SQL statement that you create. ■

Using *SELECT* Statements

The SELECT statement retrieves records (or specified fields from records) and places the information in a dynaset or table for further processing by a program. The SELECT statement follows this general form:

```
SELECT [predicate] fieldlist FROM tablelist [table relations]
     [range options] [sort options] [group options]
```

NOTE In my demonstrations of code statements, words in all caps are SQL keywords, and italicized words or phrases are used to indicate terms that a programmer would replace in an actual statement—for example, `fieldlist` would be replaced with `Lastname, Firstname`. Phrases or words inside square brackets are optional terms. ■

The various components of the preceding statement are explained in this chapter. Although a SQL statement can be greatly complex, it also can be fairly simple. The simplest form of the SELECT statement is shown here:

```
SELECT * FROM Sales
```

Defining the Desired Fields

The `fieldlist` part of the SELECT statement is used to define the fields to be included in the output recordset. You can include all fields in a table, selected fields from the table, or even calculated fields based on other fields in the table. You can also choose the fields to be included from a single table or from multiple tables.

The `fieldlist` portion of the SELECT statement takes the following form:

```
[tablename.]field1 [AS alt1][,[tablename.]field2 [AS alt2]]
```

Selecting All Fields from a Table The * wild-card parameter is used to indicate that you want to select all the fields in the specified table. The wild card is used in the `fieldlist` portion of the statement. The statement `SELECT * FROM Sales`, when used with the sample database you are developing, produces the output recordset shown in Figure B5.1.

Selecting Individual Fields from a Table Frequently, you need only a few fields from a table. You can specify the desired fields by including a field list in the SELECT statement. Within the field list, the individual fields are separated by commas. In addition, if the desired field has a space in the name, as in `Order Quantity`, the field name must be enclosed within square brackets, `[]`.The recordset that results from the following SELECT statement is shown in Figure B5.2. A recordset created with fields specified is more efficient than one created with the wild card (*), both in terms of the size of the recordset and speed of creation. As a general rule, you should limit your queries to the smallest number of fields that can accomplish your purpose.

```
SELECT [Item Code], Quantity FROM Sales
```

FIG. B5.1
Using * in the `fieldlist` parameter selects all fields from the source table.

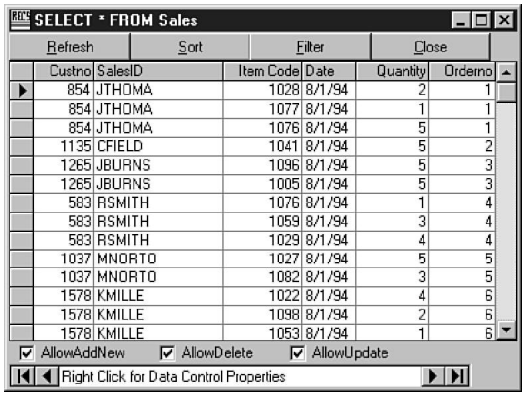
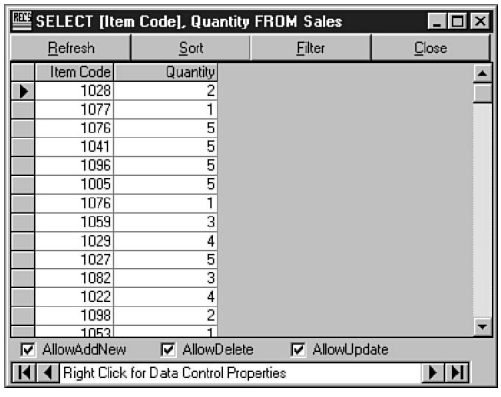


FIG. B5.2
This recordset results from specifying individual fields in the SELECT statement.



Selecting Fields from Multiple Tables As you might remember from the discussions on database design in Chapter 28, "Building Database Applications," you normalize data by placing it in different tables to eliminate data redundancy. When you retrieve this data for viewing or modification, you want to see all the information from the related tables. SQL lets you combine information from various tables into a single recordset.

▶ **See** "Designing a Database," in Chapter 28.

To select data from multiple tables, you specify three things:

- The table from which each field is selected
- The fields from which you are selecting the data
- The relationship between the tables

Specify the table for each field by placing the table name and a period in front of the field name (for example, `Sales.[Item Code]` or `Sales.Quantity`). (Remember, square brackets must enclose a field name that has a space in it.) You also can use the wild-card identifier (*) after the table name to indicate that you want all the fields from that table.

To specify the tables you're using, place multiple table names (separated by commas) in the `FROM` clause of the `SELECT` statement.

The relationship between the tables is specified either by a `WHERE` clause or by a `JOIN` condition. These elements are discussed later in this chapter.

The statement in Listing B5.1 is used to retrieve all fields from the Sales table and the Item Description and Retail fields from the Retail Items table. These tables are related by the Item Code field. Figure B5.3 shows the results of the statement.

FIG. B5.3
Selecting fields from multiple tables produces a combined recordset.

SQL Statement

Refresh | Sort | Filter | Close

Custno	SalesID	Item Code	Date	Quantity	Orderno	Item Description	Retail
854	JTHOMA	1028	8/1/94	2	1	Checker Barb	2.6
854	JTHOMA	1077	8/1/94	1	1	Black Ghost	3.5
854	JTHOMA	1076	8/1/94	5	1	Green Discus	1.6
1135	CFIELD	1041	8/1/94	5	2	Black Neon Tetra	2.35
1265	JBURNS	1096	8/1/94	5	3	Water Rose	1.55
1265	JBURNS	1005	8/1/94	5	3	Blue Gourami	1.6
583	RSMITH	1076	8/1/94	1	4	Green Discus	1.6
583	RSMITH	1059	8/1/94	3	4	Emperor Tetra	1.2
583	RSMITH	1029	8/1/94	4	4	Marbled Hatchetfish	2.65
1037	MNORTO	1027	8/1/94	5	5	Zebra Danio	1.6
1037	MNORTO	1082	8/1/94	3	5	Snakeskin Gourami	2.4
1578	KMILLE	1022	8/1/94	4	6	Striped Headstander	2.3
1578	KMILLE	1098	8/1/94	2	6	Hornwort	1.45
1578	KMILLE	1053	8/1/94	1	6	Sailfin Molly	1.85

AllowAddNew | AllowDelete | AllowUpdate

Right Click for Data Control Properties

NOTE The listing shows an underscore character at the end of each of the first three lines. This is used to break the lines for the purpose of page-width in the book. When you enter the expressions, they need to be on a single line. ■

Listing B5.1 Sales.txt—Selecting Fields from Multiple Tables in a SQL Statement

```
SELECT Sales.*, [Retail Items].[Item Description], _
    [Retail Items].Retail  _
    FROM Sales, [Retail Items]  _
    WHERE Sales.[Item Code]=[Retail Items].[Item Code]
```

NOTE You can leave out the table name when specifying fields as long as the requested field is present only in one table in the list. However, it is very good programming practice to include the table name, both for reducing the potential for errors and for readability of your code. ■

Creating Calculated Fields The example in Listing B5.1 has customer-order information consisting of the item ordered, quantity of the item, and the retail price. Suppose that you also want to access the total cost of the items. You can achieve this by using a *calculated field* in the SELECT statement. A calculated field can be the result of an arithmetic operation on numeric fields (for example, `Price * Quantity`) or the result of string operations on text fields (for example, `Lastname & Firstname`). For numeric fields, you can use any standard arithmetic operation (+, –, *, /, ^). For strings, you can use the concatenation operator (&). In addition, you can use Visual Basic functions to perform operations on the data in the fields (for example, you can use the MID$ function to extract a substring from a text field, the UCASE$ function to place text in uppercase letters, or the SQR function to calculate the square root of a number). Listing B5.2 shows how some of these functions can be used in the SELECT statement.

Listing B5.2 Totprice.txt—Creating a Variety of Calculated Fields with the *SELECT* Statement

```
'**************************************
'Calculate the total price for the items
'**************************************
SELECT [Retail Items].Retail * Sales.Quantity FROM _
       [Retail Items],Sales  _
       WHERE Sales.[Item Code]=[Retail Items].[Item Code]
'*********************************************************************
'Create a name field by concatenating the Lastname and
'Firstname fields
'*********************************************************************
SELECT Lastname & ', ' & Firstname FROM Customers
'*********************************************************************
'Create a customer ID using the first 3 letters of the Lastname
' and Firstname fields and make all letters uppercase.
'*********************************************************************
SELECT UCASE$(MID$(Lastname,1,3)) & UCASE$(MID$(Firstname,1,3))  _
   FROM Customers
'************************************************************
'Determine the square root of a number for use in a data report.
'************************************************************
SELECT Datapoint, SQR(Datapoint) FROM Labdata
```

In the listing, no field name is specified for the calculated field. The Query engine automatically assigns a name, such as `Expr1001`, for the first calculated field. The next section, "Specifying Alternative Field Names," describes how you can specify a name for the field.

Calculated fields are placed in the recordset as read-only fields—they can't be updated. In addition, if you update the base data used to create the field, the changes are not reflected in the calculated field.

NOTE If you use a calculated field with a data control, it is best to use a label control to show the contents of the field. This prevents the user from attempting to update the field and causing an error. You could also use a text box with the locked property set to `True`. (You can learn more about the Data control and bound controls by reviewing Chapters 29, "Using the Visual Basic Data Control," and 30, "Doing More with Bound Controls.") If you use a text box, you might want to change the background color to indicate to the user that the data cannot be edited. ■

Specifying Alternative Field Names Listing B5.2 created calculated fields to include in a recordset. For many applications, you will want to use a name for the field other than the one automatically created by the query engine.

You can change the syntax of the `SELECT` statement to give the calculated field a name. You assign a name by including the `AS` clause and the desired name after the definition of the field (refer to the second part of Listing B5.3). If you want, you can also use this technique to assign a different name to a standard field.

Listing B5.3 Custname.txt—Accessing a Calculated Field's Value and Naming the Field

```
'*****************************************
'Set up the SELECT statement without the name
'*****************************************
Dim NewDyn As RecordSet
SQL = "SELECT Lastname & ', ' & Firstname FROM Customers"
'*************************************
'Create a dynaset from the SQL statement
'*************************************
NewDyn = OldDB5.OpenRecordset(SQL)
'*********************************
'Get the value of the created field
'*********************************
Person = NewDyn.Recordset(0)
'*********************************************************
'Set up the SELECT statement and assign a name to the field
'*********************************************************
SQL = "SELECT Lastname & ', ' & Firstname As Name FROM Customers"
'*************************************
'Create a dynaset from the SQL statement
'*************************************
NewDyn = OldDb.OpenRecordset(SQL)
'*********************************
'Get the value of the created field
'*********************************
Person = NewDyn.Recordset("Name")
```

Specifying the Data Sources

In addition to telling the database engine what information you want, you must tell it in which table to find the information. This is done with the FROM clause of the SELECT statement. Here is the general form of the FROM clause:

```
FROM table1 [IN data1] [AS alias1][,table2 [IN data2] [AS alias2]]
```

Various options of the FROM clause are discussed in the following sections.

Specifying the Table Names The simplest form of the FROM clause is used to specify a single table. This is the form of the clause used in this statement:

```
SELECT * FROM Sales
```

The FROM clause can also be used to specify multiple tables (refer to Listing B5.1). When specifying multiple tables, separate the table names with commas. Also, if a table name has an embedded space, the table name must be enclosed in square brackets, [] (refer to Listing B5.1).

Using Tables in Other Databases As you develop more applications, you might have to pull data together from tables in different databases. For example, you might have a ZIP Code database that contains the city, state, and ZIP Code for every postal code in the United States. You do not want to have to duplicate this information in a table for each of your database applications that requires it. The SELECT statement lets you store that information once in its own database and then pull it in as needed. To retrieve the information from a database other than the current one, you use the IN portion of the FROM clause. The SELECT statement for retrieving the ZIP Code information along with the customer data is shown in Listing B5.4.

Listing B5.4 Getcust.txt—Retrieving Information from More than One Database

```
'*************************************************************
'We are working from the TRITON database which is already open.
'*************************************************************
SELECT Customers.Lastname, Customers.Firstname, Zipcode.City, _
    Zipcode.State  FROM Customers, Zipcode IN USZIPS  _
    WHERE Customers.Zip = Zipcode.Zip
```

Assigning an Alias Name to a Table Notice the way the table name for each desired field was listed in Listing B5.4. Because these table names are long and there are a number of fields, the SELECT statement is fairly long. The statement gets much more complex with each field and table you add. In addition, typing long names each time increases the chances of making a typo.

To alleviate this problem, you can assign the table an alias by using the AS portion of the FROM clause. By using AS, you can assign a unique, shorter name to each table. This alias can be used in all the other clauses in which the table name is needed. Listing B5.5 is a rewrite of the code from Listing B5.4, using the alias CS for the Customers table and ZP for the Zipcode table.

Listing B5.5 Alias.txt—Using a Table Alias to Cut Down on Typing

```
'*************************************************
'We use aliases to make the statement easier to enter.
'*************************************************
SELECT CS.Lastname, CS.Firstname, ZP.City, ZP.State  _
    FROM Customers AS CS, Zipcode IN USZIPS AS ZP   _
    WHERE CS.Zip = ZP.Zip
```

Using *ALL*, *DISTINCT*, or *DISTINCTROW* Predicates

In most applications, you select all records that meet specified criteria. You can do this by specifying the ALL predicate in front of your field names or by leaving out any predicate specification (ALL is the default behavior). Therefore, the following two statements are equivalent:

```
SELECT * FROM Customers
SELECT ALL * FROM Customers
```

Sometimes, however, you might want to determine the unique values of fields. For these times, use the DISTINCT or DISTINCTROW predicate. The DISTINCT predicate causes the database engine to retrieve only one record with a specific set of field values—no matter how many duplicates exist. For a record to be rejected by the DISTINCT predicate, its values for all the selected fields must match those of another record. For example, if you are selecting first and last names, you can retrieve several people with the last name Smith, but you can't retrieve multiple occurrences of Adam Smith.

If you want to eliminate records that are completely duplicated, use the DISTINCTROW predicate. DISTINCTROW compares the values of all fields in the table, whether or not they are among the selected fields. For the sample database, you can use DISTINCTROW to determine which products have been ordered at least once. DISTINCTROW has no effect if the query is on only a single table.

Listing B5.6 shows the uses of DISTINCT and DISTINCTROW.

Listing B5.6 Distinct.txt—Obtaining Unique Records with the DISTINCT or DISTINCTROW Predicates

```
'*****************************
'Use of the DISTINCT predicate
'*****************************
SELECT DISTINCT [Item Code] FROM Sales
```

```
'*********************************
'Use of the DISTINCTROW predicate
'*********************************
SELECT DISTINCTROW [Item Code] FROM [Retail Items], Sales  _
     [Retail Items] INNER JOIN Sales  _
     ON [Retail Items].[Item Code]=Sales.[Item Code]
```

Setting Table Relationships

When you design a database structure, you use key fields so that you can relate the tables in the database. For example, you use a salesperson ID in the Customers table to relate to the salesperson in the Salesperson table so that you don't have to include all the salesperson data with every customer record. You use these same key fields in the SELECT statement to set the table relationships so that you can display and manipulate the related data. That is, when you view customer information, you want to see the salesperson's name, not his or her ID.

You can use two clauses to specify the relationships between tables:

- JOIN This combines two tables, based on the contents of specified fields in each table and the type of JOIN.
- WHERE This usually is used to filter the records returned by a query, but it can be used to emulate an INNER JOIN. You will take a look at the INNER JOIN in the following section.

NOTE Using the WHERE clause to join tables creates a read-only recordset. To create a modifiable recordset, you must use the JOIN clause.

Using a *JOIN* Clause

The basic format of the JOIN clause is as follows:

```
table1 {INNER|LEFT|RIGHT} JOIN table2 ON table1.key1 = table2.key2
```

The Query engine used by Visual Basic (also used by Access, Excel, and other Microsoft products) supports three JOIN clauses: INNER, LEFT, and RIGHT. Each clause returns records that meet the JOIN condition, but each behaves differently in returning records that do not meet that condition. Table B5.2 shows the records returned from each table for the three JOIN conditions. For this discussion, *table1* is the left table and *table2* is the right table. In general, the left table is the first one specified (on the left side of the JOIN keyword) and the right table is the second table specified (on the right side of the JOIN keyword).

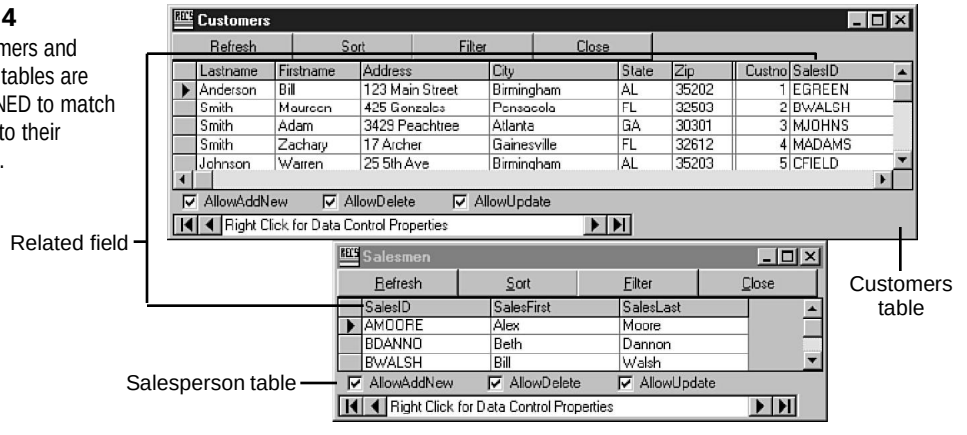
NOTE You can use any comparison operator (<, <=, =, >=, >, or <>) in the JOIN clause to relate the two tables.

Table B5.2 Records Returned Based on the Type of *JOIN* Used

JOIN Type Table	Records from Left Table	Records from Right Table
INNER	Only records with corresponding record in right table	Only records with corresponding record in left table
LEFT	All records	Only records with corresponding record in left table
RIGHT	Only records with corresponding record in right table	All records

To further understand these concepts, consider the sample database with its Customers and Salesperson tables. In that database, you created a small information set in the tables consisting of ten customers and four salespeople. Two customers have no salesperson listed, and one of the salespeople has no customers (he's a new guy). You select the same fields with each JOIN but specify an INNER JOIN, LEFT JOIN, and RIGHT JOIN (refer to Listing B5.7). Figure B5.4 shows the two base-data tables from which this listing is working. Figure B5.5 shows the resulting recordsets for each of the JOIN operations.

FIG. B5.4
The Customers and Salesmen tables are RIGHT JOINED to match salesmen to their customers.



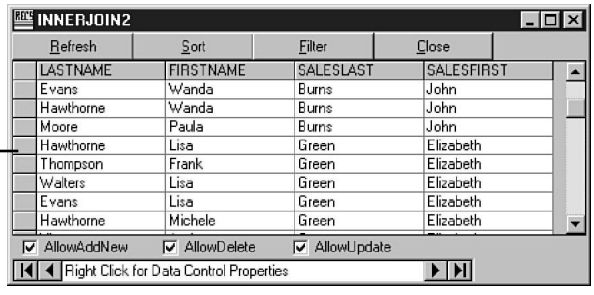
Listing B5.7 Join.txt—Examples of the Three *JOIN* Types

```
'**************************
'Select using an INNER JOIN
'**************************
SELECT CS.Lastname, CS.Firstname, SL.Saleslast, SL.Salesfirst  _
    FROM Customers AS CS, Salesmen AS SL,  _
    CS INNER JOIN SL ON CS.SalesID=SL.SalesID
```

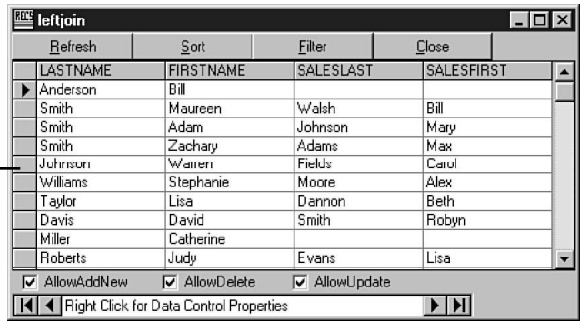
```
'*************************
'Select using an LEFT JOIN
'*************************
SELECT CS.Lastname, CS.Firstname, SL.Saleslast, SL.Salesfirst  _
    FROM Customers AS CS, Salesmen AS SL,  _
    CS LEFT JOIN SL ON CS.SalesID=SL.SalesID
'*************************
'Select using an RIGHT JOIN
'*************************
SELECT CS.Lastname, CS.Firstname, SL.Saleslast, SL.Salesfirst  _
    FROM Customers AS CS, Salesmen AS SL,  _
    CS RIGHT JOIN SL ON CS.SalesID=SL.SalesID
```

FIG. B5.5
Different records are returned with the different `JOIN` types.

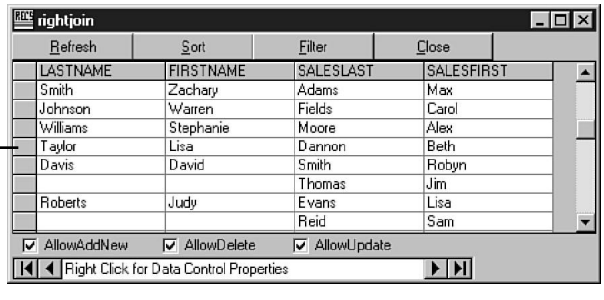
INNER JOIN



LEFT JOIN



RIGHT JOIN



Note that, in addition to returning the salesperson with no customers, the RIGHT JOIN returned all customer records for each of the other salespeople, not just a single record. This is because a RIGHT JOIN is designed to return all the records from the right table, even if they have no corresponding record in the left table.

Using the *WHERE* Clause You can use the WHERE clause to relate two tables. The WHERE clause has the same effect as an INNER JOIN. Listing B5.8 shows the same INNER JOIN as Listing B5.7, this time using the WHERE clause instead of the INNER JOIN.

Listing B5.8 Where.txt—A *WHERE* Clause Performing the Same Function as an *INNER JOIN*

```
'**************************************
'Select using WHERE to relate two tables
'**************************************
SELECT CS.Lastname, CS.Firstname, SL.Saleslast, SL.Salesfirst  _
    FROM Customers AS CS, Salesmen AS SL,  _
    WHERE CS.SalesID=SL.SalesID
```

Setting the Filter Criteria

One of the most powerful features of SQL commands is that you can control the range of records to be processed by specifying a filter condition. You can use many types of filters, such as `Lastname = "Smith"`, `Price < 1`, or `birthday between 5/1/94 and 5/31/94`. Although the current discussion is specific to the use of filters in the SELECT command, the principles shown here also work with other SQL commands, such as DELETE and UPDATE.

Filter conditions in a SQL command are specified by using the WHERE clause. The general format of the WHERE clause is as follows:

```
WHERE logical-expression
```

You can use four types of *predicates* (logical statements that define the condition) with the WHERE clause. These are shown in the following table:

Predicate	Action
Comparison	Compares a field to a given value
LIKE	Compares a field to a pattern (for example, A*)
IN	Compares a field to a list of acceptable values
BETWEEN	Compares a field to a range of values

Using the Comparison Predicate As its name suggests, the *comparison predicate* is used to compare the values of two expressions. You can use six comparison operators (the symbols that describe the comparison type); the operators and their definitions are summarized in Table B5.3.

Table B5.3 Comparison Operators Used in the *WHERE* Clause

Operator	Definition
<	Less than
<=	Less than or equal to
=	Equal to
>=	Greater than or equal to
>	Greater than
<>	Not equal to

Here is the generic format of the comparison predicate:

```
expression1 comparison-operator expression2
```

For all comparisons, both expressions must be of the same type (for example, both must be numbers or both must be text strings). Several comparisons of different types are shown in Listing B5.9. The comparison values for strings and dates require special formatting. Any strings used in a comparison must be enclosed in single quotes (for example, `'Smith'` or `'AL'`). Likewise, dates must be enclosed between pound signs (for example, `#5/15/94#`). The quotes and the pound signs tell the Query engine the type of data that is being passed. Note that numbers do not need to be enclosed within special characters.

Listing B5.9 Compare.txt—Comparison Operators Used with Many Types of Data

```
'******************************************************
'Comparison of text data using customer table as source
'******************************************************
SELECT * FROM Customers WHERE Lastname='Smith'
'************************************************
'Comparison of numeric data using Retail Items table
'************************************************
SELECT * FROM [Retail Items] WHERE Retail<2
'****************************************
'Comparison of date data using Sales table
'****************************************
SELECT * FROM Sales WHERE Date>#8/15/94#
```

Using the *LIKE* Predicate With the `LIKE` predicate, you can compare an *expression* (that is, a field value) to a pattern. The `LIKE` predicate lets you make comparisons such as last names starting with *S*, titles containing *SQL*, or five-letter words starting with *M* and ending with *H*. You use the wild cards `*` and `?` to create the patterns. The actual predicates for these comparisons would be `Lastname LIKE 'S*'`, `Titles LIKE '*SQL*'`, and `Word LIKE 'M???H'`, respectively.

The LIKE predicate is used exclusively for string comparisons. The format of the LIKE predicate is as follows:

```
expression LIKE pattern
```

The patterns defined for the LIKE predicate make use of wild-card matching and character-range lists. When you create a pattern, you can combine some of the wild cards and character lists to allow greater flexibility in the pattern definition. When used, character lists must meet three criteria:

- The list must be enclosed within square brackets.
- The first and last characters must be separated by a hyphen.
- The range of the characters must be defined in ascending order (for example, a–z, and not z–a).

In addition to using a character list to match a character in the list, you can precede the list with an exclamation point to indicate that you want to exclude the characters in the list. Table B5.4 shows the type of pattern matching you can perform with the LIKE predicate. Listing B5.10 shows the use of the LIKE predicate in several SELECT statements.

Table B5.4 The *LIKE* Predicate Using a Variety of Pattern Matching

Wild Card	Used to Match	Example Pattern	Example Results
*	Multiple characters	S*	Smith, Sims, sheep
?	Single character	an?	and, ant, any
#	Single digit	3524#	35242, 35243
[list]	Single character in list	[c-f]	d, e, f
[!list]	Single character not in list	[!c-f]	a, b, g, h
combination	Specific to pattern	a?t*	art, antique, artist

Listing B5.10 Like.txt—Use the *LIKE* Predicate for Pattern Matching

```
'***************************
'Multiple character wild card
'***************************
SELECT * FROM Customers WHERE Lastname LIKE 'S*'
'************************
'Single character wild card
'************************
SELECT * FROM Customers WHERE State LIKE '?L'
'**********************
```

```
'Character list matching
'**********************
SELECT * FROM Customers WHERE MID$(Lastname,1,1) LIKE '[a-f]'
```

Using the *IN* Predicate The IN predicate lets you determine whether the expression is one of several values. With the IN predicate, you can check state codes for customers to determine whether the customer's state matches a sales region. This example is shown in the following sample code:

```
SELECT * FROM Customers WHERE State IN ('AL', 'FL', 'GA')
```

Using the *BETWEEN* Predicate The BETWEEN predicate lets you search for expressions with values within a range of values. You can use the BETWEEN predicate for string, numeric, or date expressions. The BETWEEN predicate performs an *inclusive search*, meaning that if the value is equal to one of the endpoints of the range, the record is included. You can also use the NOT operator to return records outside the range. The form of the BETWEEN predicate is as follows:

```
expression [NOT] BETWEEN value1 AND value2
```

Listing B5.11 shows the use of the BETWEEN predicate in several scenarios.

Listing B5.11 Between.txt—Using the *BETWEEN* Predicate to Check an Expression Against a Range of Values

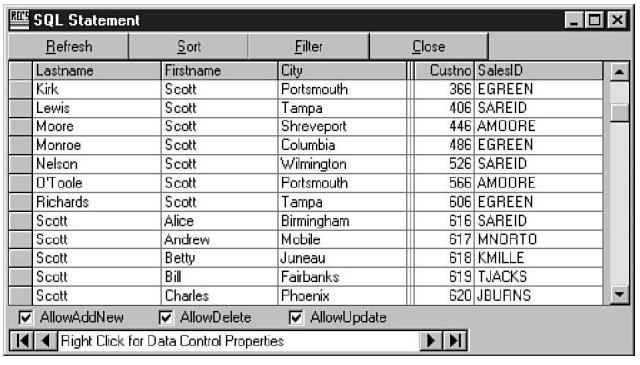
```
'******************
'String comparison
'******************
SELECT * FROM Customers WHERE Lastname BETWEEN 'M' AND 'W'
'******************
'Numeric comparison
'******************
SELECT * FROM [Retail Items] WHERE Retail BETWEEN 1 AND 2.5
'***************
'Date comparison
'***************
SELECT * FROM Sales WHERE Date BETWEEN #8/01/94# AND #8/10/94#
'************************
'Use of the NOT operator
'************************
SELECT * FROM Customers WHERE Lastname NOT BETWEEN 'M' AND 'W'
```

Combining Multiple Conditions The WHERE clause can also accept multiple conditions so that you can specify filtering criteria on more than one field. Each individual condition of the multiple conditions is in the form of the conditions described in the preceding sections on using predicates. These individual conditions are then combined by using the logical operators AND and OR. By using multiple-condition statements, you can find all the Smiths in the Southeast, or you can find anyone whose first or last name is Scott. Listing B5.12 shows the statements for these examples. Figure B5.6 shows the recordset resulting from a query search for Scott.

Listing B5.12 Andor.txt—Combining Multiple *WHERE* Conditions with *AND* or *OR*

```
'*********************************
'Find all Smiths in the Southeast
'*********************************
SELECT * FROM Customers WHERE Lastname = 'Smith' AND   _
    State IN ('AL', 'FL', 'GA')
'************************************************
'Find all occurrences of Scott in first or last name
'************************************************
SELECT * FROM Customers WHERE Lastname = 'Scott' _
     OR Firstname = 'Scott'
```

FIG. B5.6
You can use multiple conditions to enhance a WHERE clause.



Setting the Sort Conditions

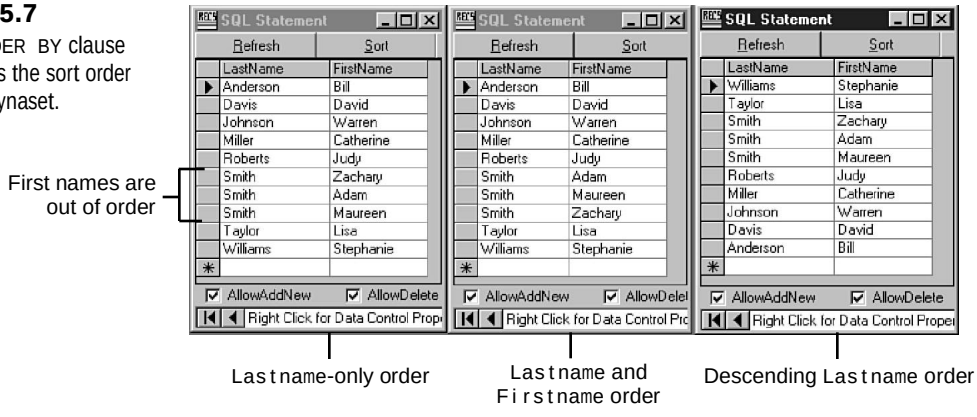
In addition to specifying the range of records to process, you can also use the SELECT statement to specify the order in which you want the records to appear in the output dynaset. The SELECT statement controls the order in which the records are processed or viewed. Sorting the records is done by using the ORDER BY clause of the SELECT statement.

You can specify the sort order with a single field or with multiple fields. If you use multiple fields, the individual fields must be separated by commas.

The default sort order for all fields is ascending (that is, A–Z, 0–9). You can change the sort order for any individual field by specifying the DESC keyword after the field name (the DESC keyword affects only the one field, not any other fields in the ORDER BY clause). Listing B5.13 shows several uses of the ORDER BY clause. Figure B5.7 shows the results of these SELECT statements.

NOTE When you're sorting records, the presence of an index for the sort field can significantly speed up the SQL query.

FIG. B5.7
The ORDER BY clause specifies the sort order of the dynaset.



Listing B5.13 Sort.txt—Specifying the Sort Order of the Output Dynaset

```
'*****************
'Single field sort
'*****************
SELECT * FROM Customers ORDER BY Lastname
'*******************
'Multiple field sort
'*******************
SELECT * FROM Customers ORDER BY Lastname, Firstname
'*********************
'Descending order sort
'*********************
SELECT * FROM Customers ORDER BY Lastname DESC, Firstname
```

Using Aggregate Functions

You can use the SELECT statement to perform calculations on the information in your tables by using the SQL *aggregate functions*. To perform the calculations, define them as a field in your SELECT statement, using the following syntax:

```
function(expression)
```

The expression can be a single field or a calculation based on one or more fields, such as `Quantity * Price` or `SQR(Datapoint)`. The `Count` function can also use the wild card * as the expression, because `Count` returns only the number of records. Table B5.5 shows the 11 aggregate functions available in Microsoft SQL.

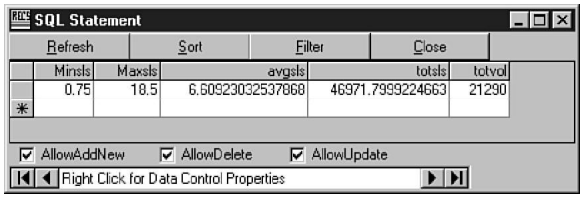
Table B5.5 Aggregate Functions Provide Summary Information About Data in the Database

Function	Returns
Avg	The arithmetic average of the field for the records that meet the WHERE clause
Count	The number of records that meet the WHERE clause
Min	The minimum value of the field for the records that meet the WHERE clause
Max	The maximum value of the field for the records that meet the WHERE clause
Sum	The total value of the field for the records that meet the WHERE clause
First	The value of the field for the first record in the recordset
Last	The value of the field for the last record in the recordset
StDev	The standard deviation of the values of the field for the records that meet the WHERE clause
StDevP	The standard deviation of the values of the field for the records that meet the WHERE clause
Var	The variance of the values of the field for the records that meet the WHERE clause
VarP	The variance of the values of the field for the records that meet the WHERE clause

NOTE In Table B5.5, StDev and StDevP seem to perform the same function. The same is true of Var and VarP. The difference between the functions is that the StDevP and VarP evaluate populations where StDev and Var evaluate samples of populations. ■

As with other SQL functions, these aggregate functions operate only on the records that meet the filter criteria specified in the WHERE clause. Aggregate functions are unaffected by sort order. Aggregate functions return a single value for the entire recordset unless the GROUP BY clause (described in the following section) is used. If GROUP BY is used, a value is returned for each record group. Listing B5.14 shows the SELECT statement used to calculate the minimum, maximum, average, and total sales amounts, as well as the total item volume from the Sales table in the sample case. Figure B5.8 shows the output from this query.

FIG. B5.8
The table shows the summary information from aggregate functions.



Listing B5.14 Summary.txt—Using Aggregate Functions to Provide Summary Information

```
SELECT Min(SL.Quantity * RT.Retail) AS Minsls,  _
     Max(SL.Quantity * RT.Retail) AS Maxsls,  _
     Avg(SL.Quantity * RT.Retail) AS Avgsls,  _
     Sum(SL.Quantity * RT.Retail) AS Totsls,  _
     Sum(SL.Quantity) AS Totvol  _
     FROM Sales AS SL, [Retail Items] AS RT  _
     WHERE SL.[Item Code]=RT.[Item Code]
```

Creating Record Groups

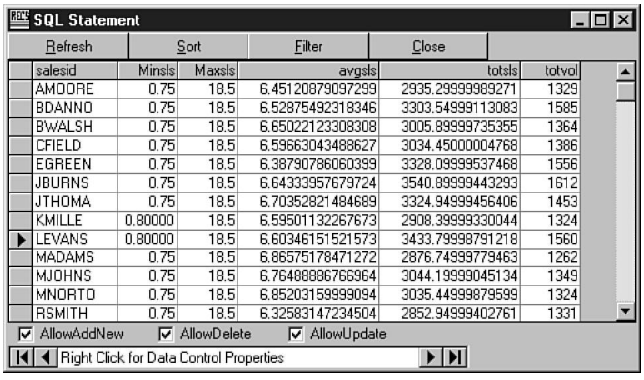
Creating record groups lets you create a recordset that has only one record for each occurrence of the specified field. For example, if your group the Customers table by state, you have one output record for each state. This arrangement is especially useful when combined with the calculation functions described in the preceding sections. When groups are used with aggregate functions, you can easily obtain summary data by state, salesperson, item code, or any other desired field.

Most of the time, you want to create groups based on a single field. You can, however, specify multiple fields in the `GROUP BY` clause. If you do, a record is returned for each unique combination of field values. You can use this technique to get sales data by salesperson and item code. Separate multiple fields in a `GROUP BY` clause with commas. Listing B5.15 shows an update of Listing B5.14, adding groups based on the salesperson ID. Figure B5.9 shows the results of the query.

Listing B5.15 Group.txt—Using the *GROUP BY* Clause to Obtain Summary Information for Record Groups

```
SELECT SL.SalesID, Min(SL.Quantity * RT.Retail) AS Minsls,  _
     Max(SL.Quantity * RT.Retail) AS Maxsls,  _
     Avg(SL.Quantity * RT.Retail) AS Avgsls,  _
     Sum(SL.Quantity * RT.Retail) AS Totsls,  _
     Sum(SL.Quantity) AS Totvol  _
     FROM Sales AS SL, [Retail Items] AS RT  _
     WHERE SL.[Item Code]=RT.[Item Code]  _
     GROUP BY SL.SalesID
```

FIG. B5.9
Using GROUP BY creates a summary record for each defined group.



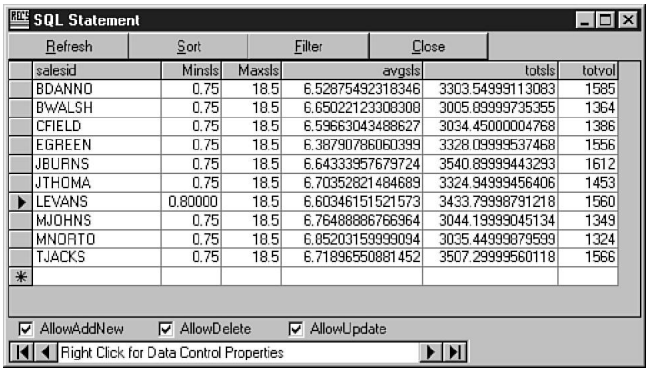
salesid	Minsls	Maxsls	avgsls	totsls	totvol
AMOORE	0.75	18.5	6.45120879097299	2935.29999989271	1329
BDANNO	0.75	18.5	6.52875492318346	3303.54999113083	1585
BWALSH	0.75	18.5	6.65022123308308	3005.89999735355	1364
CFIELD	0.75	18.5	6.59663043488627	3034.45000004768	1386
EGREEN	0.75	18.5	6.38790786060399	3328.09999537468	1556
JBURNS	0.75	18.5	6.64333957679724	3540.89999443293	1612
JTHOMA	0.75	18.5	6.70352821484689	3324.94999456406	1453
KMILLE	0.80000	18.5	6.59501132267673	2908.39999330044	1324
LEVANS	0.80000	18.5	6.60346151521573	3433.79998791218	1560
MADAMS	0.75	18.5	6.86575178471272	2876.74999779463	1262
MJOHNS	0.75	18.5	6.76488886766964	3044.19999045134	1349
MNORTO	0.75	18.5	6.85203159999094	3035.44999879599	1324
RSMITH	0.75	18.5	6.32583147234504	2852.94999402761	1331

The GROUP BY clause can also include an optional HAVING clause. The HAVING clause works similarly to a WHERE clause but examines only the field values of the returned records. The HAVING clause determines which of the selected records to display; the WHERE clause determines which records to select from the base tables. You can use the HAVING clause to display only those salespeople with total sales exceeding $3,000 for the month. Listing B5.16 shows this example; Figure B5.10 shows the output from this listing.

Listing B5.16 Having.txt—The *HAVING* Clause Filters the Display of the Selected Group Records

```
SELECT SL.SalesID, Min(SL.Quantity * RT.Retail) AS Minsls,  _
     Max(SL.Quantity * RT.Retail) AS Maxsls,  _
     Avg(SL.Quantity * RT.Retail) AS Avgsls,  _
     Sum(SL.Quantity * RT.Retail) AS Totsls,  _
     Sum(SL.Quantity) AS Totvol  _
     FROM Sales AS SL, [Retail Items] AS RT  _
     SL INNER JOIN RT ON SL.[Item Code]=RT.[Item Code]  _
     GROUP BY SL.SalesID  _
     HAVING Sum(SL.Quantity * RT.Retail) > 3000
```

FIG. B5.10
The HAVING clause limits the display of group records.



salesid	Minsls	Maxsls	avgsls	totsls	totvol
BDANNO	0.75	18.5	6.52875492318346	3303.54999113083	1585
BWALSH	0.75	18.5	6.65022123308308	3005.89999735355	1364
CFIELD	0.75	18.5	6.59663043488627	3034.45000004768	1386
EGREEN	0.75	18.5	6.38790786060399	3328.09999537468	1556
JBURNS	0.75	18.5	6.64333957679724	3540.89999443293	1612
JTHOMA	0.75	18.5	6.70352821484689	3324.94999456406	1453
LEVANS	0.80000	18.5	6.60346151521573	3433.79998791218	1560
MJOHNS	0.75	18.5	6.76488886766964	3044.19999045134	1349
MNORTO	0.75	18.5	6.85203159999094	3035.44999879599	1324
TJACKS	0.75	18.5	6.71896550881452	3507.29999560118	1566

Creating a Table

In all the examples of the SELECT statement used earlier in this chapter, the results of the query were output to a dynaset or a snapshot. Because these recordsets are only temporary, their contents exist only as long as the recordset is open. After a close method is used or the application is terminated, the recordset disappears (although any changes made to the underlying tables are permanent).

Sometimes, however, you might want to permanently store the information in the recordset for later use. Do so with the INTO clause of the SELECT statement. With the INTO clause, you specify the name of an output table (and, optionally, the database for the table) in which to store the results. You might want to do this to generate a mailing-list table from your customer list. This mailing-list table can then be accessed by your word processor to perform a mail-merge function or to print mailing labels. Listing B5.4, earlier in this chapter, generated such a list in a dynaset. Listing B5.17 shows the same basic SELECT statement as was used in Listing B5.4, but with the INTO clause used to store the information in a table.

Listing B5.17 Into.txt—Using the INTO Clause to Save Information to a New Table

```
SELECT CS.Firstname & ' ' & CS.Lastname, CS.Address, ZP.City, _
    ZP.State, CS.ZIP INTO Mailings FROM Customers AS CS, _
    Zipcode IN USZIPS AS ZP WHERE CS.Zip = ZP.Zip
```

CAUTION

The table name you specify should be that of a new table. If you specify the name of a table that already exists, that table is overwritten with the output of the SELECT statement.

Using Parameters

So far in all the clauses, you have seen specific values specified. For example, you specified 'AL' for a state and $1.25 for a price. But what if you don't know in advance what value you want to use in comparison? Well, this is precisely what parameters are used for in a SQL statement. The parameter is to the SQL statement what a variable is to a program statement. The parameter is a placeholder whose value is assigned by your program before the SQL statement is executed.

To use a parameter in your SQL statement, you first have to specify the parameter in the PARAMETERS declaration part of the statement. The PARAMETERS declaration comes before the SELECT or other manipulative clause in the SQL statement. The declaration specifies both the name of the parameter and its data type. The PARAMETERS clause is separated from the rest of the SQL statement by a semicolon.

After you have declared the parameters, you simply place them in the manipulative part of the statement where you want to be able to substitute a value. The following code line shows how a parameter would be used in place of a state ID in a SQL statement:

```
PARAMETERS StateName String; SELECT * FROM Customers
    WHERE State = StateName
```

When you go to run the SQL statement in your program, each parameter is treated like a property of the `QueryDef`. Therefore, you need to assign a value to each parameter before you use the `Execute` method. The following code shows you how to set the property value for the preceding SQL statement and open a recordset:

```
Dim OldDb As Database, Qry As QueryDef, Rset As Recordset
Set OldDb = DBEngine.Workspaces(0).OpenDatabase("C:\Triton.Mdb")
Set Qry = OldDb.QueryDefs("StateSelect")
Qry!StateName = "AL"
Set Rset = Qry.OpenRecordset()
```

As you can see, using parameters makes it easy to store your queries in the database and still maintain the flexibility of being able to specify comparison values at run time.

SQL Action Statements

In the previous section, you saw how the `SELECT` statement can be used to retrieve records and place the information in a dynaset or table for further processing by a program. This was just one of the four manipulative statements that you defined earlier in this chapter. The three remaining statements are as follows:

- `DELETE FROM` An action query that removes records from a table
- `INSERT INTO` An action query that adds a group of records to a table
- `UPDATE` An action query that sets the values of fields in a table

In the following sections, you look at how to use these statements to further refine that data that you are manipulating in a database via a SQL function.

Using the *DELETE* Statement

The `DELETE` statement is used to create an *action query.* The `DELETE` statement's purpose is to delete specific records from a table. An action query does not return a group of records into a dynaset as `SELECT` queries do. Instead, action queries work like program *subroutines.* That is, an action query performs its functions and returns to the next statement in the calling program.

The syntax of the `DELETE` statement is as follows:

```
DELETE FROM tablename [WHERE clause]
```

The `WHERE` clause is an optional parameter. If it is omitted, all the records in the target table are deleted. You can use the `WHERE` clause to limit the deletions to only those records that meet

specified criteria. In the WHERE clause, you can use any of the comparison predicates defined in the earlier section "Using the Comparison Predicate." Following is an example of the DELETE statement used to eliminate all customers who live in Florida:

```
DELETE FROM Customers WHERE State='FL'
```

CAUTION

After the DELETE statement is executed, the records are gone and can't be recovered. The only exception is if transaction processing is used. If you're using transaction processing, you can use a ROLLBACK statement to recover any deletions made since the last BEGINTRANS statement was issued.

Using the *INSERT* Statement

Like the DELETE statement, the INSERT statement is another action query. The INSERT statement is used with the SELECT statement to add a group of records to a table. The syntax of the statement is as follows:

```
INSERT INTO tablename SELECT rest-of-select-statement
```

You build the SELECT portion of the statement exactly as explained in the first part of this chapter in the section "Using *SELECT* Statements." The purpose of the SELECT portion of the statement is to define the records to be added to the table. The INSERT statement defines the action of adding the records and specifies the table that is to receive the records.

One use of the INSERT statement is to update tables created with the SELECT INTO statement. Suppose that you're keeping a church directory. When you first create the directory, you create a mailing list for the current member list. Each month, as new members are added, you either can rerun the SELECT INTO query and re-create the table, or you can run the INSERT INTO query and add only the new members to the existing mailing list. Listing B5.18 shows the creation of the original mailing list and the use of the INSERT INTO query to update the list.

Listing B5.18 Insert.txt—Using the *INSERT INTO* Statement to Add a Group of Records to a Table

```
'*******************************
'Create a new mailing list table
'*******************************
SELECT CS.Firstname & ' ' & CS.Lastname, CS.Address, ZP.City, _
    ZP.State, CS.ZIP INTO Mailings FROM Members AS CS, _
    Zipcode IN USZIPS AS ZP WHERE CS.Zip = ZP.Zip
'*********************************
'Update the mailing list each month
'*********************************
INSERT INTO Mailings SELECT CS.Firstname & ' ' & CS.Lastname,  _
     CS.Address, ZP.City, ZP.State, CS.ZIP  _
     FROM Customers AS CS, Zipcode IN USZIPS AS ZP   _
     WHERE CS.Zip = ZP.Zip AND CS.Memdate>Lastmonth
```

Using the *UPDATE* Statement

The UPDATE statement is another action query. It is used to change the values of specific fields in a table. The syntax of the UPDATE statement is as follows:

```
UPDATE tablename SET field = newvalue [WHERE clause]
```

You can update multiple fields in a table at one time by listing multiple `field = newvalue` clauses, separated by commas. The inclusion of the WHERE clause is optional. If it is excluded, all records in the table are changed.

Listing B5.19 shows two examples of the UPDATE statement. The first example changes the salesperson ID for a group of customers, as happens when a salesperson leaves the company and his or her accounts are transferred to someone else. The second example changes the retail price of all retail sales items, as can be necessary to cover increased operating costs.

Listing B5.19 Update.txt—Using the *UPDATE* Statement to Change Field Values for Many Records at Once

```
'*****************************************
'Change the SalesID for a group of customers
'*****************************************
UPDATE Customers SET SalesID = 'EGREEN' WHERE SalesID='JBURNS'
'*****************************************************
'Increase the retail price of all items by five percent
'*****************************************************
UPDATE [Retail Items] SET Retail = Retail * 1.05
```

Using Data-Definition-Language Statements

Data-definition-language statements (DDLs) let you create, modify, and delete tables and indexes in a database with a single statement. For many situations, these statements take the place of the data-access-object methods described in Chapter 28, "Building Database Applications." However, there are some limitations to using the DDL statements. The main limitation is that these statements are supported only for Jet databases (remember that data-access objects can be used for any database accessed with the Jet engine). The other limitation of DDL statements is that they support only a small subset of the properties of the table, field, and index objects. If you need to specify properties outside of this subset, you must use the methods described in Chapter 28.

Defining Tables with DDL Statements

Three DDL statements are used to define tables in a database:

- CREATE TABLE Defines a new table in a database
- ALTER TABLE Changes a table's structure
- DROP TABLE Deletes a table from the database

Creating a Table with DDL Statements To create a table with the DDL statements, you create a SQL statement containing the name of the table and the names, types, and sizes of each field in the table. The following code shows how to create the Orders table of the sample case:

```
CREATE TABLE Orders (Orderno LONG, Custno LONG, SalesID TEXT (6), _
    OrderDate DATE, Totcost SINGLE)
```

Notice that when you specify the table name and field names, you do not have to enclose the names in quotation marks. However, if you want to specify a name with a space in it, you must enclose the name in square brackets (for example, `[Last name]`).

When you create a table, you can specify only the field names, types, and sizes. You can't specify optional parameters such as default values, validation rules, or validation error messages. Even with this limitation, the DDL `CREATE TABLE` statement is a powerful tool that you can use to create many of the tables in a database.

Modifying a Table By using the `ALTER TABLE` statement, you can add a field to an existing table or delete a field from the table. When adding a field, you must specify the name, type, and (when applicable) the size of the field. You add a field by using the `ADD COLUMN` clause of the `ALTER TABLE` statement. To delete a field, you need to specify only the field name and use the `DROP COLUMN` clause of the statement. As with other database-modification methods, you can't delete a field used in an index or a relation. Listing B5.20 shows how to add and then delete a field from the Orders table created in the preceding section.

Listing B5.20 Altertab.txt—Using the *ALTER TABLE* Statement to Add or Delete a Field from a Table

```
'************************************************
'Add a shipping charges field to the "Orders" table
'************************************************
ALTER TABLE Orders ADD COLUMN Shipping SINGLE
'*********************************
'Delete the shipping charges field
'*********************************
ALTER TABLE Orders DROP COLUMN Shipping
```

Deleting a Table You can delete a table from a database by using the `DROP TABLE` statement. The following simple piece of code shows how to get rid of the Orders table. Use caution when deleting a table; the table and all its data are gone forever after the command has been executed.

```
DROP TABLE Orders
```

Defining Indexes with DDL Statements

Two DDL statements are designed especially for use with indexes:

- `CREATE INDEX` Defines a new index for a table
- `DROP INDEX` Deletes an index from a table

Creating an Index You can create a single-field or multi-field index with the `CREATE INDEX` statement. To create the index, you must give the name of the index, the name of the table for the index, and at least one field to be included in the index. You can specify ascending or descending order for each field. You can also specify that the index is a primary index for the table. Listing B5.21 shows how to create a primary index on customer number and a two-field index with the sort orders specified. These indexes are set up for the Customers table of the sample case.

Listing B5.21 Createind.txt—Create Several Types of Indexes with the *CREATE INDEX* Statement

```
'****************************************
'Create a primary index on customer number
'****************************************
CREATE INDEX Custno ON Customers (Custno) WITH PRIMARY
'************************************************************
'Create a two field index with ascending order on Lastname and
'   descending order on Firstname.
'************************************************************
CREATE INDEX Name2 ON Customers (Lastname ASC, Firstname DESC)
```

Deleting an Index Getting rid of an index is just as easy as creating one. To delete an index from a table, use the `DROP INDEX` statement as shown in the following example. These statements delete the two indexes created in Listing B5.21. Notice that you must specify the table name for the index that you want to delete:

```
DROP INDEX Custno ON Customers
DROP INDEX Name2 ON Customers
```

Using SQL

As stated at the beginning of the chapter, you can't place a SQL statement by itself in a program. It must be part of another function. This part of the chapter describes the various methods used to implement the SQL statements.

Executing an Action Query

The Jet engine provides an execute method as part of the database object. The execute method tells the engine to process the SQL query against the database. An action query can be executed by specifying the SQL statement as part of the execute method for a database. An action query can also be used to create a `QueryDef`. Then the query can be executed on its own. Listing B5.22 shows how both of these methods are used to execute the same SQL statement.

Listing B5.22 Execute.txt—Run SQL Statements with the *DatabaseExecute* or *QueryExecute* Method

```
Dim OldDb AS Database, NewQry AS QueryDef
'*************************************************
'Define the SQL statement and assign it to a variable
'*************************************************
SQLstate = "UPDATE Customers SET SalesID = 'EGREEN'"
SQLstate = SQLstate + " WHERE SalesID='JBURNS'"
'***************************************
'Use the database execute to run the query
'***************************************
OldDb.Execute SQLstate
'*************************************
'Create a QueryDef from the SQL statement
'*************************************
Set NewQry = OldDb.CreateQueryDef("Change Sales", SQLstate)
'*********************************
'Use the query execute to run the query
'*********************************
NewQry.Execute
'*************************************************
'Run the named query with the database execute method
'*************************************************
OldDb.Execute "Change Sales"
```

Creating a *QueryDef*

Creating a `QueryDef` lets you name your query and store it in the database with your tables. You can create either an action query or a *retrieval query* (one that uses the `SELECT` statement). After the query is created, you can call it by name for execution (shown in a listing in the previous section "Executing an Action Query") or for creation of a dynaset (as described in the following section). Listing B5.22 showed how to create a `QueryDef` called `Change Sales` that is used to update the salesperson ID for a group of customers.

Creating Dynasets and Snapshots

To use the `SELECT` statement to retrieve records and store them in a dynaset or snapshot, you must use the `SELECT` statement with the `OpenRecordset` method. By using the `OpenRecordset` method, you specify the type of recordset with the options parameter. With this method, you either can use the `SELECT` statement directly or use the name of a retrieval query that you have previously defined. Listing B5.23 shows these two methods of retrieving records.

Listing B5.23 Createmeth.txt—Using the Create Methods to Retrieve the Records Defined by a *SELECT* Statement

```
Dim OldDb As Database, NewQry As QueryDef, NewDyn As Recordset
Dim NewSnap As Recordset
'*****************************************************
'Define the SELECT statement and store it to a variable
'*****************************************************
SQLstate = "SELECT RI.[Item Description], SL.Quantity,"
SQLstate = SQLstate & " RI.Retail, _
     SL.Quantity * RI.Retail AS Subtot"
SQLstate = SQLstate & "FROM [Retail Items] AS RI, Sales AS SL"
SQLstate = SQLstate & "WHERE SL.[Item Code]=RI.[Item Code]"
'************************
'Create dynaset directly
'************************
Set NewDyn = OldDb.OpenRecordset(SQLstate, dbOpenDynaset)
'***************
'Create QueryDef
'***************
Set NewQry = OldDb.CreateQueryDef("Get Subtotals", SQLstate)
NewQry.Close
'*****************************
'Create snapshot from querydef
'*****************************
Set NewSnap = OldDb.OpenRecordset("Get Subtotals", dbOpenSnapshot)
```

You have seen how SELECT statements are used to create dynasets and snapshots. But, the comparison part of a WHERE clause and the sort list of an ORDER BY clause can also be used to set dynaset properties. The filter property of a dynaset is a WHERE statement without the WHERE keyword. When setting the filter property, you can use all the predicates described earlier in the section "Using the *WHERE* Clause." In a like manner, the sort property of a dynaset is an ORDER BY clause without the ORDER BY keywords.

Using SQL Statements with the Data Control

The data control uses the RecordSource property to create a recordset when the control is loaded. The RecordSource can be a table, a SELECT statement, or a predefined query. Therefore, the entire discussion on the SELECT statement (in the section "Using *SELECT* statements") applies to the creation of the recordset used with a data control.

NOTE When you specify a table name for the RecordSource property, Visual Basic uses the name to create a SELECT statement such as this:

```
SELECT * FROM table
```

Creating SQL Statements

When you create and test your SQL statements, you can program them directly into your code and run the code to see whether they work. This process can be very time-consuming and frustrating, especially for complex statements. However, three easier ways of developing SQL statements might be available to you:

- The Visual Data Manager add-in that comes with Visual Basic
- Microsoft Access (if you have a copy)
- Microsoft Query

NOTE Users of Microsoft Excel or Microsoft Office also have access to Microsoft Query, the tool in Access.

The Visual Data Manager and Access both have query builders that can help you create SQL queries. They provide dialog boxes for selecting the fields to include, and they help you with the various clauses. When you have finished testing a query with either application, you can store the query as a `QueryDef` in the database. This query can then be executed by name from your program. As an alternative, you can copy the code from the query builder into your program, using standard cut-and-paste operations.

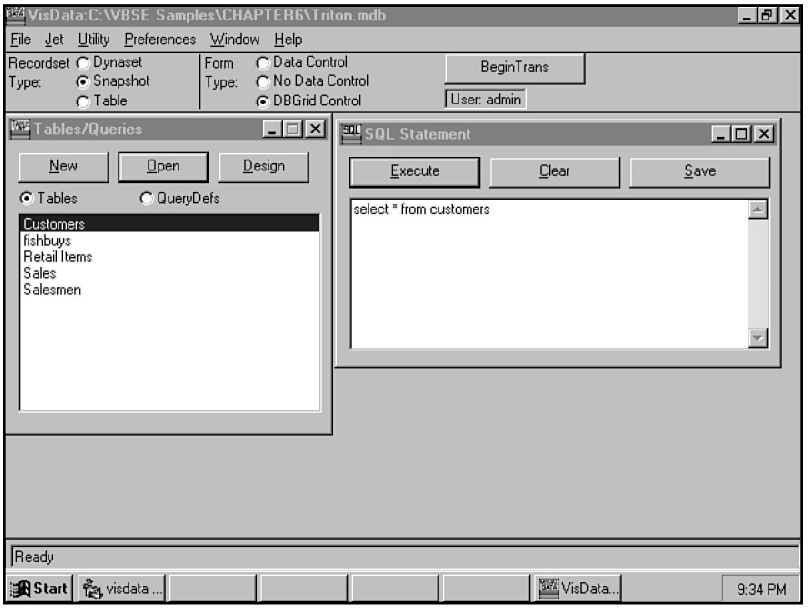
Using the Visual Data Manager

The Visual Data Manager is a Visual Basic add-in that allows you to create and modify databases for your Visual Basic programs. The Visual Data Manager also has a window that allows you to enter and debug SQL queries. And if you don't want to try to create the query yourself, VDM has a query builder that makes it easy for you to create queries by making choices in the builder.

NOTE If you want to learn about the inner workings of the Visual Data Manager, it is one of the sample projects installed with Visual Basic. The project file is VISDATA.VBP and is found in the VISDATA folder of the Samples folder.

To start the Visual Data Manager, simply select the Visual Data Manager item from the Add-Ins menu of Visual Basic. After starting the program, open the File menu and select the Open Database item; then select the type of database to open from the submenu. You are presented with a dialog box that allows you to open a database. After the database is opened, a list of the tables and queries in the database appears in the left window of the application. The Visual Data Manager with the Triton.Mdb database open is shown in Figure B5.11.

FIG. B5.11
You can use the Visual Data Manager Add-In to develop SQL queries.

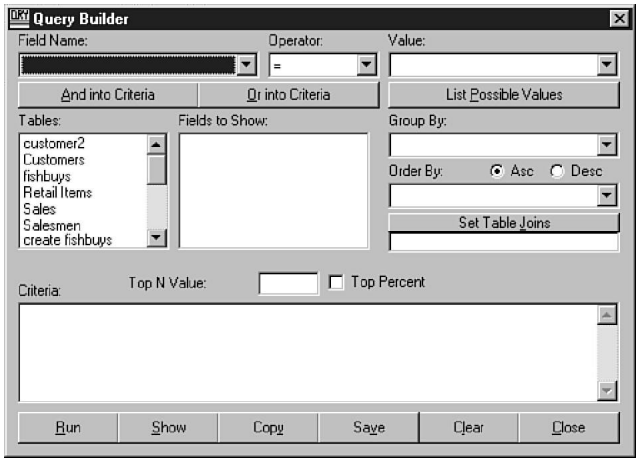


To develop and test SQL statements, first enter the statement in the text box of the SQL dialog box (the one on the right of Figure B5.11). When you're ready to test the statement, click the Execute SQL button. If you're developing a retrieval query, a dynaset is created and the results are displayed in a data entry form (or a grid) if the statement has no errors. If you're developing an action query, a message box appears, telling you that the execution of the query is complete (again, assuming that the statement is correct). If you have an error in your statement, a message box appears informing you of the error.

The Visual Data Manager add-in also includes a Query Builder. You can access the Query Builder (shown in Figure B5.12) by choosing Query Builder from the Utilities menu of the Visual Data Manager. To create a query with the Query Builder, follow these steps:

1. Select the tables to include from the Tables list.
2. Select the fields to include from the Fields to Show list.
3. Set the `WHERE` clause (if any) using the Field Name, Operator, and Value drop-down lists at the top of the dialog box.
4. Set the table `JOIN` conditions (if any) by clicking the Set Table Joins command button.
5. Set a single-field `ORDER BY` clause (if any) by selecting the field from the Order By Field drop-down box and selecting either the Asc or Desc option.
6. Set a single `GROUP BY` field (if any) by selecting the field from the Group By Field drop-down box.

FIG. B5.12
The Query Builder makes it easy to build SQL statements.



After you have set the Query Builder parameters, you can run the query, display the SQL statement, or copy the query to the SQL statement window. The Query Builder provides an easy way to become familiar with constructing `SELECT` queries.

When you have developed the query to your satisfaction (either with the Query Builder or by typing the statement directly), you can save the query as a `QueryDef` in your database. In your Visual Basic code, you can then reference the name of the query you created. Alternatively, you can copy the query from Visual Data Manager and paste it into your application code.

Using Microsoft Access

If you have a copy of Microsoft Access, you can use its query builder to graphically construct queries. You can then save the query as a `QueryDef` in the database and reference the query name in your Visual Basic code.

One of more creative uses for Access is to reverse-engineer a `QueryDef`. Microsoft Access allows you to build a graphical representation of the tables and databases for a particular `QueryDef` entered in SQL format. This reverse-engineering process gives you a unique way to debug or make modifications graphically to an existing query.

Optimizing SQL Performance

Developers always want to get the best possible performance from every aspect of their applications. Wanting high performance out of SQL queries is no exception. Fortunately, you can use several methods to optimize the performance of your SQL queries.

Using Indexes

The Microsoft Jet database engine uses an optimization technology called Rushmore. Under certain conditions, Rushmore uses available indexes to try to speed up queries. To take maximum advantage of this arrangement, you can create an index on each of the fields you typically use in a WHERE clause or a JOIN condition. This is particularly true of key fields used to relate tables (for example, the Custno and SalesID fields in the sample database). An index also works better with comparison operators than with the other types of WHERE conditions, such as LIKE or IN.

NOTE Only certain types of queries are optimizable by Rushmore. For a query to use Rushmore optimization, the WHERE condition must use an indexed field. In addition, if you use the LIKE operator, the expression should begin with a character, not a wild card. Rushmore works with Jet databases and FoxPro and dBase tables. Rushmore does not work with ODBC databases. ■

Compiling Queries

Compiling a query refers to creating a QueryDef and storing it in the database. If the query already exists in the database, the command parser does not have to generate the query each time it is run, and this increases execution speed. If you have a query that is frequently used, create a QueryDef for it.

Keeping Queries Simple

When you're working with a lot of data from a large number of tables, the SQL statements can become quite complex. Complex statements are much slower to execute than simple ones. Also, if you have a number of conditions in WHERE clauses, this increases complexity and slows execution time.

Keep statements as simple as possible. If you have a complex statement, consider breaking it into multiple smaller operations. For example, if you have a complex JOIN of three tables, you might be able to use the SELECT INTO statement to create a temporary table from two of the three and then use a second SELECT statement to perform the final JOIN. There are no hard-and-fast rules for how many tables are too many or how many conditions make a statement too complex. If you're having performance problems, try some different ideas and find the one that works best.

Another way to keep things simple is to try to avoid pattern matching in a WHERE clause. Because pattern matching does not deal with discrete values, it is hard to optimize. In addition, patterns that use wild cards for the first character are much slower than those that specifically define that character. For example, if you're looking for books about SQL, finding ones with *SQL* anywhere in the title (pattern = "*SQL*") requires looking at every title in the table. On the other hand, looking for titles that start with *SQL* (pattern = "SQL*") lets you skip over most records. If you had a Title index, the search would go directly to the first book on SQL.

Passing SQL Statements to Other Database Engines

Visual Basic can pass a SQL statement through to an ODBC database server such as SQL Server. When you pass a statement through, the Jet engine does not try to do any processing of the query, but it sends the query to the server to be processed. Remember, however, that the SQL statement must conform to the SQL syntax of the host database.

To use the pass-through capability, set the options parameter in the `OpenRecordset` or the execute methods to the value of the `dbSQLPassThrough` constant.

NOTE The project file, SQLDEMO.VBP, also on the companion CD, contains many of the listings used in this chapter. Each listing is assigned to a command button. Choosing the command button creates a dynaset with the SQL statement in the listing and displays the results in a data-bound grid. The form containing the grid also has a text box that shows the SQL statement. ■

From Here...

This chapter has taught you the basics of using SQL in your database program. You have seen how to select records and how to limit the selection by using the `WHERE` clause. You have also seen how SQL statements can be used to modify the structure of a database and how to use aggregate functions to obtain summary information.

To see how SQL statements are used in programs and with the data control, refer to the following chapters:

- Chapter 28, "Building Database Applications," explains how to write data-access programs.
- Chapters 29, "Using the Visual Basic Data Control," and 30, "Doing More with Bound Controls," explain how to use the Data control.