CHAPTER 27

Building Queries in an Access Project

As you already know from Chapter 7, "Creating and Working with Simple Queries," Chapter 8, "Building Complex Queries," and Chapter 9, "Modifying Data with Action Queries," you can quickly build nearly all the queries that you need in an Access desktop database (.accdb) using the query designer. The query designer in an Access project file (.adp) offers an interface that is similar to the one you use in an Access desktop database. So, with only a small learning curve, you can soon be building queries in your Microsoft Office Access 2007 projects.

Remember that your Access project file doesn't store your data or your queries. You must connect your project to Microsoft SQL Server 2005 Express Edition or Microsoft SQL Server, and it's the server that contains your data and queries. SQL Server offers query capabilities that go far beyond the already robust features of an Access desktop database (.accdb). SQL Server offers several different types of queries that allow you to tap into the power potential of SQL Server. You can build many of the queries you need for your application using the query designer. However, to use the full potential of SQL Server, you'll need to learn its programming language, Transact-SQL, and build queries in the text editor. Transact-SQL not only includes SQL statements that are compliant with the ANSI SQL-92 intermediate standard but also offers extended capabilities that make it more like a programming language wrapped around the existing query language.

Note

You can find a reference guide to the SQL supported by Office Access 2007 and SQL Server in Article 2, "Understanding SQL," on the companion CD. To obtain a complete reference to Transact-SQL, download a free copy of the Microsoft *SQL Server 2005 Books Online* (a set of help files) from *www.microsoft.com/technet/prodtechnol/sql/2005/downloads/books.mspx*.

Viewing Queries in an Access Project

When you open a project file that is properly connected to an SQL Server or SQL Server 2005 Express Edition database, you can view the queries in the database by clicking the Navigation Pane menu and clicking Object Type under Navigate To Category and then clicking Queries under Filter By Group, as shown in Figure 27-1. Although the queries appear to be stored directly in your project file, they're actually in the database on the server.

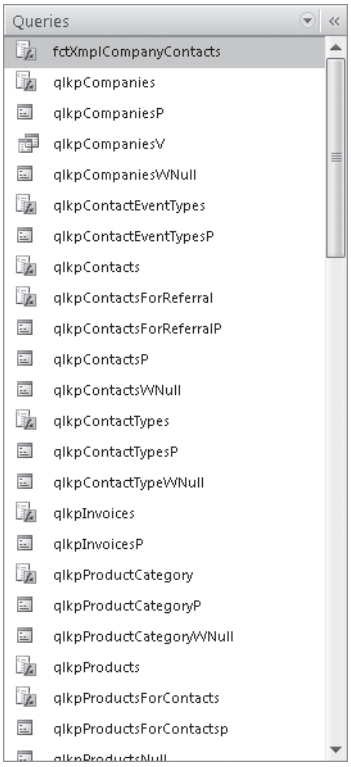

Figure 27-1 This is the list of queries in the ContactsSQL.mdf sample database as seen from the Contacts.adp sample project file.

Note

To follow along with the examples in this chapter, you can open the Contacts.adp sample project file, or you can create a new project file and connect it to the ContactsSQL.mdf sample database. You can find both files on your companion CD. See Chapter 26, "Building Tables in an Access Project," for details about how to create a new project file and connect it to an existing database. You can also connect a new project to the Contact-Tracking database that you created in Chapter 26.


View

In an Access project file, you can work with three different types of objects in SQL Server as queries. A *view* most closely resembles select queries that you can create in an Access desktop database—a logical table that returns one or more columns from one or more tables. The major differences between a view in an Access project (.adp) and a SELECT statement in an Access desktop database (.accdb) are that views cannot accept parameters, and you can include a sorting specification (ORDER BY) only when you also include the TOP keyword in the SELECT clause.


Function

A *function* can return a table (an *in-line function*), a set of columns from multiple tables (a *table-valued function*), or a single calculated value (a *scalar function*). You can call functions from other functions, from views, or from stored procedures (described next). You can use an in-line function as the record source for a form or report or the row source for a combo box or a list box. The one major restriction in a function is it cannot call any non-deterministic function—a function that subsequently returns a different value even when passed the same set of parameters. For example, the GETDATE built-in function is non-deterministic because it always returns a different value depending on the setting of your system clock. Of course, a function can accept parameters.


Stored Procedure

A *stored procedure* is arguably the most powerful programming object that you can create in SQL Server. With a stored procedure, you can define the equivalent of a desktop select query, a select query with parameters, or an action (UPDATE, INSERT, DELETE, or SELECT INTO) query. You can also use the full capabilities of the Transact-SQL language to define complex sets of actions to perform when your code executes the stored procedure. When a stored procedure returns a logical table, you can use it as the record source in a form or report or the row source in a combo box or list box. If the stored procedure returns a logical table as the result of executing multiple SELECT statements, the table is "read-only." When your stored procedure returns a logical table from a single SELECT statement, you can use it as the record source in an updatable form.

> **Note**
>
> You can include filters within a stored procedure based on parameters, and you can set the parameter values using properties in a form or report bound to a stored procedure. However, you cannot apply a filter to a form or report based on a stored procedure—not with the filter commands on the Ribbon, not by setting the Filter and FilterOn properties in code, and not by including a WhereCondition parameter on an OpenForm or OpenReport command in code.

The object of this chapter is to familiarize you with the query design facilities in an Access project file. The chapter is not intended to teach you everything there is to know about Transact-SQL—that would require another entire book. However, you should be able to take what you learn in this chapter and apply the design techniques that you learned in the three query design chapters for desktop databases and build most of the queries that you need for an application built with a project file.

The first section of this chapter, "Building Queries Using the Query Designer," explains step by step how to build the types of queries you are already familiar with. You can use the query designer to create views, in-line functions, and stored procedures that return a logical table. The second main section, "Building Queries Using a Text Editor," shows you how to get started building more complex queries using Transact-SQL. In the text editor, you can create complex stored procedures, scalar functions, and table-valued functions.

INSIDE OUT

Additional Steps You Must Take to Be Able to Work with the Sample Project File

To be able to use the Conrad Systems Contacts sample project file (Contacts.adp) and its associated SQL Server database (ContactsSQL.mdf), you must first either install SQL Server 2005 Express Edition or have permission to create databases on SQL Server version 2000 or later on your network. You can download a copy of SQL Server 2005 Express Edition from *http://msdn2.microsoft.com/en-us/sql/aa336346.aspx*—see the Appendix, "Installing Your Software," for details.

The simplest method is to install and start SQL Server 2005 Express Edition on your computer. When you first open the Contacts.adp sample project file, Visual Basic code executes and attempts to connect to your local server and find the ContactsSQL database used by the project. If it can't find the database on your local server, the code gives you the option to open the Data Link Properties dialog box to connect the database files to your server from the default sample file installation location. See "Connecting to an Existing SQL Server Database" on page 1452 for details. If you have installed the sample files in a different location or have moved the sample database files, this method might not work reliably.

If you have previously used the Microsoft SQL Server Data Engine (MSDE), you should note that Microsoft does not support running MSDE on the Windows Vista operating system. If you are running Windows Vista on your computer, you should download a copy of SQL Server 2005 Express Edition. All examples in this chapter were created using SQL Server 2005.

Another method is to use the sample command files provided on the companion CD to connect the database to your local server before you open the sample project for the first time. You'll find four small files in the \Microsoft Press\Access 2007 Inside Out\SQL folder to help you attach the sample database files to and detach the sample database files from your local server. You can also find files to shrink (compact) the sample database when it is connected to your server. Three of the files are .bat batch files that you can run by double-clicking them in Windows Explorer or by typing their names at a command prompt. (To run the files in Windows Vista, you should right-click the batch file that you want to run and click Run As Administrator on the shortcut menu.) The Attach Contacts.bat file connects the ContactsSQL.mdf and ContactsSQL.ldf files to your server, and the Detach Contacts.bat file disconnects them (a good precaution to follow before moving or updating the database files from outside Access). But you might have to do a little more preparatory work before you use either of these files.

If you view the Attach Contacts.bat file, you'll see it contains only two lines:

```
OSQL -E -i "C:\Microsoft Press\Access 2007 Inside Out\SQL\Attach Contacts.SQL"
Pause
```

The batch file invokes OSQL, a command-line program that lets you run SQL commands, which then calls an SQL script file, Attach Contacts.SQL. The second line pauses the execution so that you can see any return messages from the server. The SQL script also consists of only one line, this time an SQL command:

```
sp_attach_db "ContactsSQL", "C:\Microsoft Press\Access 2007 Inside Out\SQL\
ContactsSQL.mdf", "C:\Microsoft Press\Access 2007 Inside Out\SQL\ContactsSQL.ldf"
```

The Detach Contacts.bat file also runs the OSQL program, but it points to the Detach Contacts.SQL file that contains an sp_detach_db command.

In order to work correctly as written, all four files must be located in your Access 2007 Inside Out\SQL folder (installed under C:\Microsoft Press), and your database files must be located in the same subfolder and be named ContactsSQL.mdf and ContactsSQL.ldf. You can easily adapt these files, however, by opening each file in a plain-text editor such as Notepad and carefully adjusting the database names and paths as necessary. After adjusting the files as necessary, run the Attach Contacts.bat file by locating it in Windows Explorer and double-clicking it. You should now be able to open the Contacts.adp sample project file with no problems.

Building Queries Using the Query Designer

Particularly if you are already familiar with building queries in a desktop database, using the query designer in an Access project is the best way to get started designing queries for SQL Server.

Understanding the Query Designer

As noted earlier, you can use the query designer to create three different types of queries—views, in-line functions, and stored procedures that return a result from a single SELECT statement. Although these queries have different purposes, most of the tools you use to create them in Design view are the same. Let's start by looking at the common elements in the query designer for all query types. Later sections of this chapter cover the differences for each query type.

Adding Tables, Views, and Functions

Let's start by creating a new view. Start Access, and open the Contacts.adp sample project file or a project file that you have connected to the ContactsSQL.mdf sample database. Click the Query Wizard button in the Other group on the Create tab. Access opens the New Query dialog box, as shown in Figure 27-2.

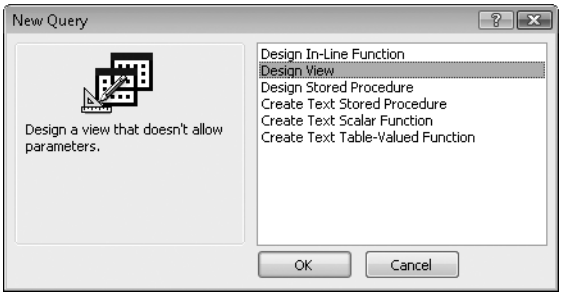

Figure 27-2 The New Query dialog box is your starting point to create new views, functions, and stored procedures.

You can see that this dialog box gives you options to begin designing in-line functions, views, and stored procedures in the query designer. The remaining three types of queries are the ones that you can build in the text editor—text stored procedures, text scalar functions, and text table-valued functions. (We'll discuss these last three options in "Building Queries Using a Text Editor" later in this chapter.) To begin creating a new view, select Design View in the New Query dialog box, and then click OK. Access opens a blank view in the query designer window and displays the Add Table dialog box shown in Figure 27-3.

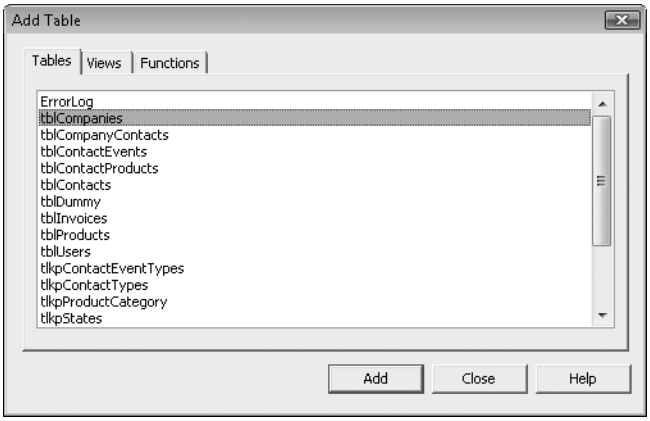

Figure 27-3 You can add tables, views, and functions to a new query using the Add Table dialog box.

The Add Table dialog box allows you to specify which tables, views, and functions you can use as a record source for your new query. You can switch between the different types of record sources by clicking the tabs at the top of the window. Double-click a record source name to add that record source to the design grid. You can also click a name to select it, hold down the Ctrl key and click multiple noncontiguous names to select them, or hold down the Shift key and select a range of names. After you have selected all the record sources you want, click the Add button to add them to your new query.

Go ahead and add the tblCompanies table to this view, and click Close.

Understanding the Panes in the Query Designer

After you add the tblCompanies table and close the Add Table dialog box, you see a query design window, as shown in Figure 27-4. By default, the query design window shows two panes: the diagram pane at the top of the window and the grid pane below it.

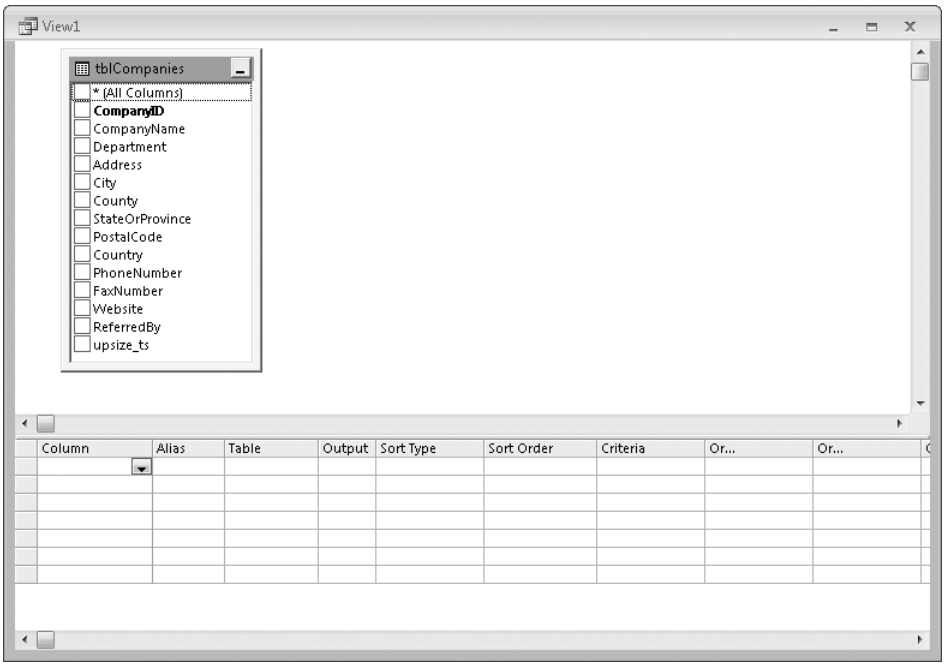

Figure 27-4 A new view in the query designer is shown with the tblCompanies table added.

The diagram pane shows a graphical representation of the tables, functions, and views that you are using to create the query and the relationships between them. The grid pane shows you information about the columns you are using in the query, similar to the design grid for a query in a desktop database. However, the query designer in a project lists the columns down instead of across. For each column, you can see the column name, any alias you assign to the column, the source table of the column, an indication that the column will be output by the query (similar to the Show check box in a desktop database query design grid), and the sorting and filtering criteria for the column.

When you're working in a desktop database, you must switch to SQL view to see the SQL that defines the query. However, in a project file, you can view and edit the SQL by opening the SQL pane. Click the SQL button in the Tools group on the Design tab under Function & View Tools to open the SQL pane, as shown in Figure 27-5.

INSIDE OUT

Use the SQL Pane to Learn How to Write SQL

One way to learn more about building queries using SQL is to always display the SQL pane when you're building a query. Each time you add or delete a record source, add or delete a relationship in the diagram pane, or change specifications in the design grid, Access updates the SQL pane to reflect your changes.

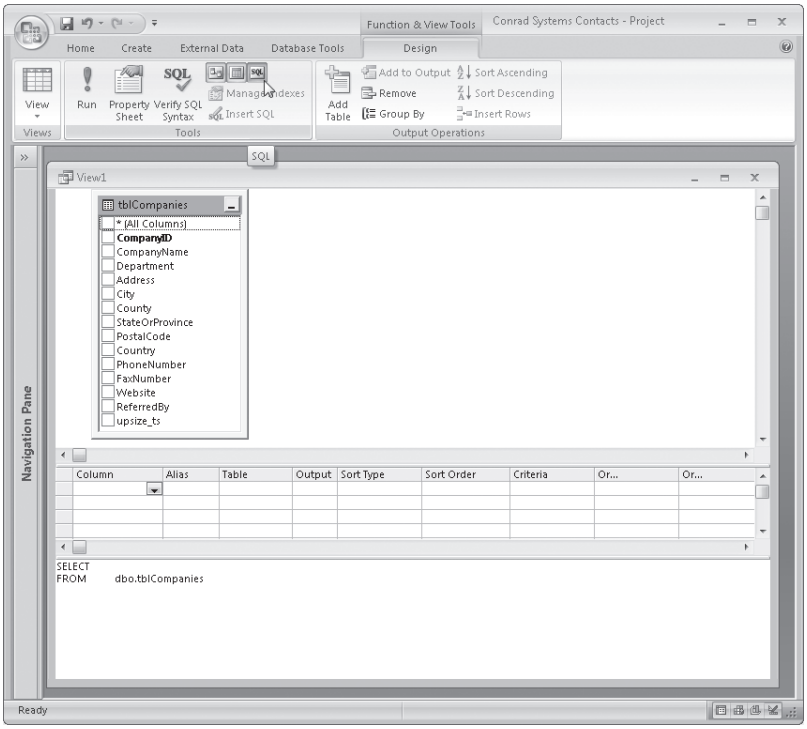

Figure 27-5 Click the SQL button to display the SQL pane while working in the query designer window.

You can enter an SQL statement directly in the SQL pane, and the query designer updates the display as soon as you move the focus away from the SQL pane (by clicking in the grid pane or the diagram pane). However, the query designer can graphically represent only a small subset of the possible SQL statements you might use. If you enter SQL code that the designer cannot display, you receive a warning message stating that the code cannot be represented graphically. The next time you open the query in Design view, you will see the SQL-only text version.

Selecting Columns

You can add columns to your query in several ways. You can drag a column name from the diagram pane and drop it on the grid pane. You can also add a column by clicking the box to the left of the column name in the diagram pane. When you do this, a check mark appears next to the column name, and Access adds the column to the grid pane, as shown in Figure 27-6.

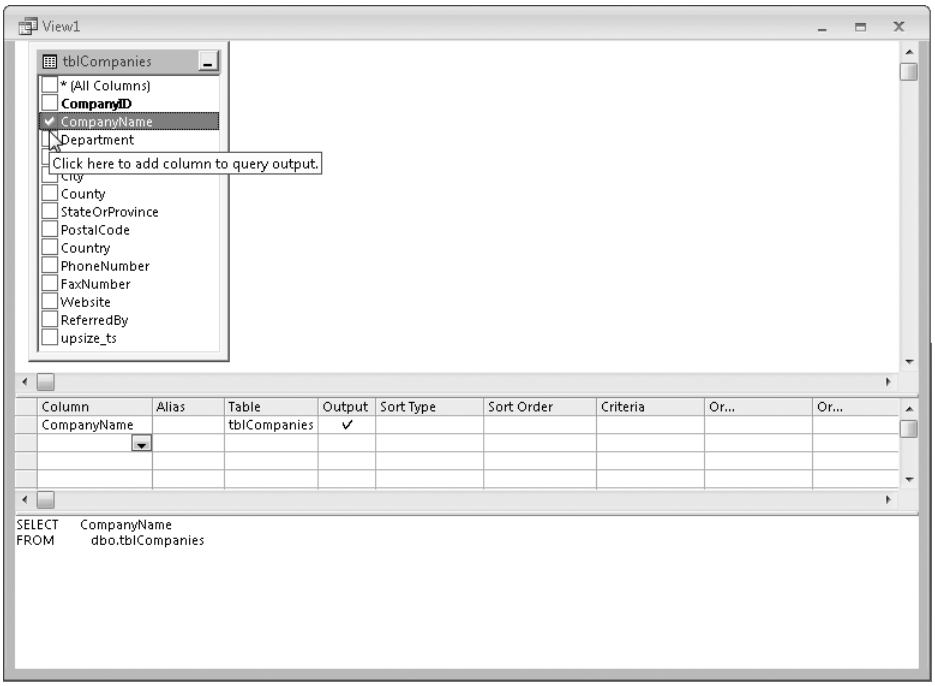

Figure 27-6 You can add a column to the view by placing a check mark next to the column name in the diagram pane.

You can also select a column from the Column list in the grid pane, as shown in Figure 27-7. If you have specified multiple record sources in the diagram pane, you can filter the list of available column names in the Column list by first choosing the record source from the Table list. Add the CompanyName and Department columns to the view by clicking next to the column listings in the diagram pane.

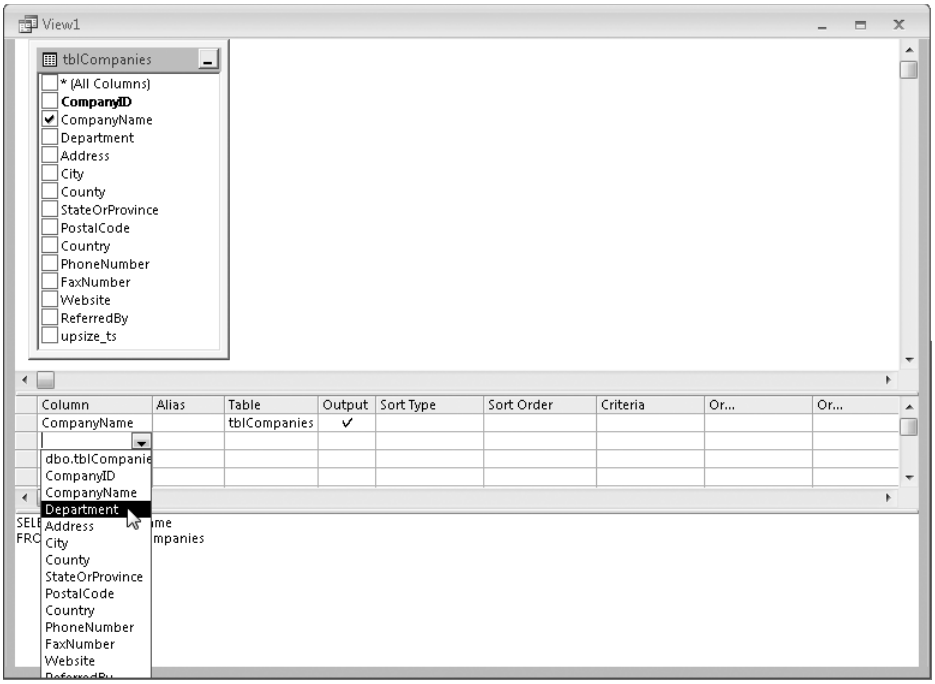

Figure 27-7 You can also add a column to a view by selecting the column name from the list in the grid pane.

Working in the Diagram Pane

After you have started working on the query, you can add other record sources by right-clicking the blank area of the diagram pane and clicking Add Table on the shortcut menu. You can also click the Add Table button in the Output Operations group on the Design tab. Access displays the Add Table dialog box so that you can select additional record sources (Figure 27-3). You can remove any record source from the diagram pane by right-clicking the record source and selecting Remove or by clicking the title bar of the record source to select it and pressing the Delete key. Add the tblCompanyContacts and tblContacts tables using the Add Table dialog box. Click Close when you are done. Your design grid should now look like Figure 27-8.

If a relationship exists between two tables in the diagram pane, Access automatically creates an inner join and displays a line connecting the primary key of one table (the key symbol) to the foreign key of the related table (the infinity symbol), similar to the way such a relationship is represented in a desktop database diagram (as discussed in Chapter 26). Remember from Chapter 8 that an inner join returns only the rows from the two tables that have matching values in both tables. Note in Figure 27-8 the two INNER JOIN clauses that Access created (in the SQL pane) and the lines connecting the three tables (in the diagram pane) that represent the defined relationships.

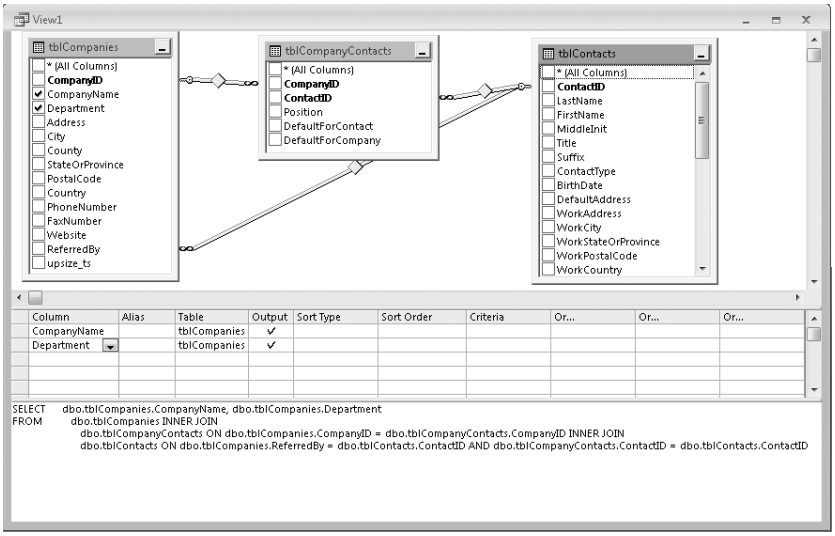

Figure 27-8 The design grid for your new view shows the tblCompanyContacts and tblContacts tables added.

When you include views or functions in your query, Access creates a join for you if it can find fields with the same name and data type in two of the record sources. You can create additional joins by clicking one column name in a record source and dragging and dropping it onto the related column name in a separate record source. To remove a join, click it once to select it, and then press the Delete key. You can also remove a join by right-clicking it and clicking Remove on the shortcut menu. Because the tblContacts table is related both to the tblCompanyContacts table and to the tblCompanies table, the query designer shows an extra relationship. For this query, you don't need the relationship between the ContactID column in the tblContacts table and the ReferredBy column in the tblCompanies table. Click this join line, and press Delete to remove it.

You can specify the type of join you want by right-clicking the line representing the join between the two tables. On the shortcut menu, you can click an option to select all rows from either table in the join. This is the same as specifying a left or a right join when writing the query in SQL. (Notice that Access updates the syntax in the SQL pane to say LEFT JOIN or RIGHT JOIN if you select one of these two options.) You can specify additional options for the join by viewing the join properties. Right-click the join between the tblCompanies and tblCompanyContacts tables, and click Properties. Access displays the Properties window, as shown in Figure 27-9.

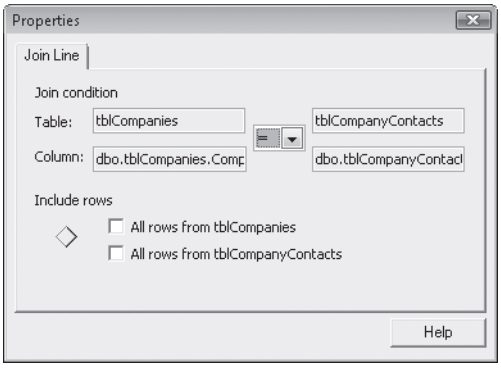

Figure 27-9 You can specify the join line properties between the tblCompanies table and the tblCompanyContacts table in the Properties window.

INSIDE OUT

Left, Right, and Full Outer Join—What's the Difference?

When you specify an outer join in the FROM clause of an SQL statement, you're asking the database to return all the rows from one or both record sources that participate in the join, regardless of whether the rows in one record source have matching rows in the other record source. A left or right outer join returns all the rows from only one of the two record sources. *Left* or *right* specifies whether the record source on the *left* (the first one in the join statement when reading left to right) or the one on the *right* is the one that returns all its rows. Access adds record source names to the FROM clause in the sequence in which you add the record sources to the diagram pane. You can later move record sources around in the diagram pane, but the first one you added still remains the one on the left in the SQL statement.

As you learned in Chapter 8, you can modify the join properties in a query in a desktop database to return all the rows from one table or the other. When you do this, Access modifies the SQL behind the scenes to use a left or right join, as appropriate. You cannot create a query in a desktop database that returns all rows from both tables.

However, SQL Server supports returning all rows from both record sources, and when you modify the join properties to ask for this, Access changes the SQL to specify a full outer join. When you run a full outer join query, you might see some rows that contain column values from the first record source and null values in the columns from the second record source, some rows that contain column values from both record sources, and some rows that contain null values in the columns from the first record source and column values from the second record source.

The top section of the Properties window allows you to specify an operator for the join condition. The default operator, an equal sign, asks the database to return rows only where the values in the column on one side of the join exactly match the values in the

joined column of the other table. However, you can use a different operator if you want the query to return rows that are unmatched (<>), greater than (>), greater than or equal to (>=), less than (<), or less than or equal to (<=) the corresponding rows in the joined table. If you specify anything other than an equal join (also known as an *equi-join*) between the two tables, the diamond-shaped symbol on the join line in the diagram pane displays that operator within the diamond symbol. The diamond symbol in the diagram pane remains blank for an equal join.

The bottom section of the Properties window allows you to specify whether you want all rows returned from one or both of the tables. Doing so is the same as right-clicking the join line in the diagram pane and clicking one of the Select All Rows From... options. Select the All Rows From tblCompanies check box. Notice that the diamond-shaped symbol on the left side of the window changes, as shown in Figure 27-10. One side of it is now box-shaped (similar to the shape of home plate on a baseball field). The diamond on the join line in the diagram pane also changes with the box side of the shape facing toward the tblCompanies table. This graphical representation means that all records are being selected from the tblCompanies table (an outer join).

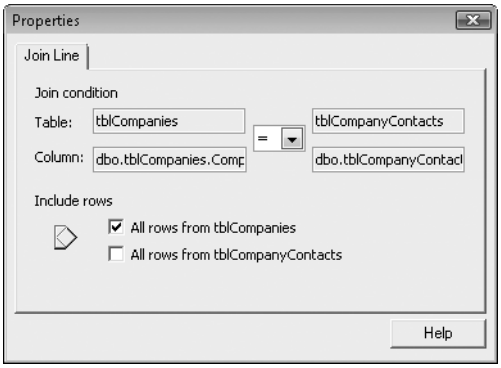

Figure 27-10 Adjust the join line properties between the tblCompanies and tblCompanyContacts tables by selecting the All Rows From tblCompanies check box to change the join to an outer join.

If you modify the SQL to specify a left or a right join between the two tables, the diamond symbol changes to a box shape on the side of the join line that connects to the table returning all rows. If you ask for a full outer join in the SQL, the entire diamond symbol becomes a box. Clear the All Rows From tblCompanies check box, and close the Properties window.

Working in the Grid Pane

You can specify a number of options for each column in the grid pane. In the Alias field, you can assign a different name to the column when it is output by the query. If you specified a Caption property for the column in the table when you created it, the query displays the caption as the heading of the column when you open the query in Datasheet view. However, the column name or the alias you specify is still the correct name to use when referring to the query column from other queries and in forms and reports.

In addition to selecting a column name from one of the record sources, you can enter an expression for a column. The expression can be as simple as a literal value or can be a complex mathematical or concatenation expression using columns from the record sources. When you enter an expression for a column, you must also enter an alias name to assign a name to the output column. If you do not enter an alias, Access generates an alias name as ExprN, where *N* is an integer value starting with 1 for the first generated alias, 2 for the second, and so on.

The Sort Type option allows you to specify how you want a column to be sorted in the query results. If you request a sort on multiple columns, you can indicate the sort order for each column to rank the order in which the database performs the sort. For example, assume you want to sort this view by department and then by company name. In the grid pane, select Ascending for both the Department and CompanyName columns from the Sort Type list. Because you want Department sorted first, be sure to choose a value of 1 in the Sort Order list. Your view design should now look like Figure 27-11. Also note that when you specify a sort on a column, an A-Z sort icon appears next to the column name in the diagram pane with an arrow indicating in which direction the sort is being applied.

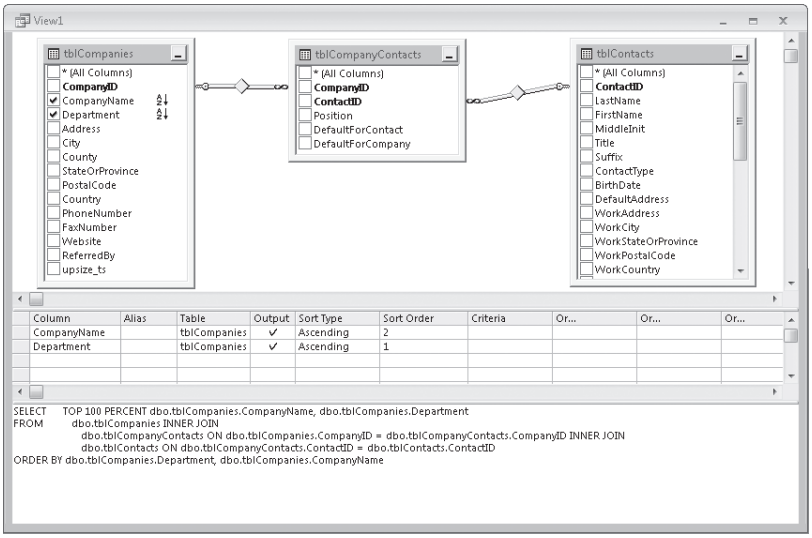

Figure 27-11 Specify a sort type and sort order for the CompanyName and Department columns.

If you want to change the order in which the columns appear in the query results, in the grid pane click the selection button to the left of the column name you want to move and drag it up or down the grid pane to move it to the desired location.

> **Note**
>
> Remember, changing the order in which the columns are displayed in a result set does not change the order in which they are sorted (unlike in an Access desktop database).

Use the Criteria field to enter criteria you want to apply to the column in the query. You can enter criteria exactly as you would in a query in an Access desktop database, except you must use single quotes to surround character and date/time literals, and you must use the % and _ wildcards in LIKE criteria instead of * and ?, respectively. (See Chapters 7 and 8 for details about specifying criteria in a query.) You can use the Or fields to specify additional criteria. Entering criteria in separate Or fields is the same as using the OR Boolean operator between two criteria statements.

You can convert any query in the query designer into an aggregate query (one that groups, totals, and otherwise performs math on the results) by clicking the Group By button in the Output Operations group on the Design tab. This is identical to clicking the Totals button in the Show/Hide group on the Design tab under Query Tools when you are working with queries in an Access desktop database, but you have many more aggregate function options in an Access project.

After you click the Group By button, Access displays an additional column in the grid pane that allows you to choose the different aggregate functions that you can apply to your columns, as shown in Figure 27-12.

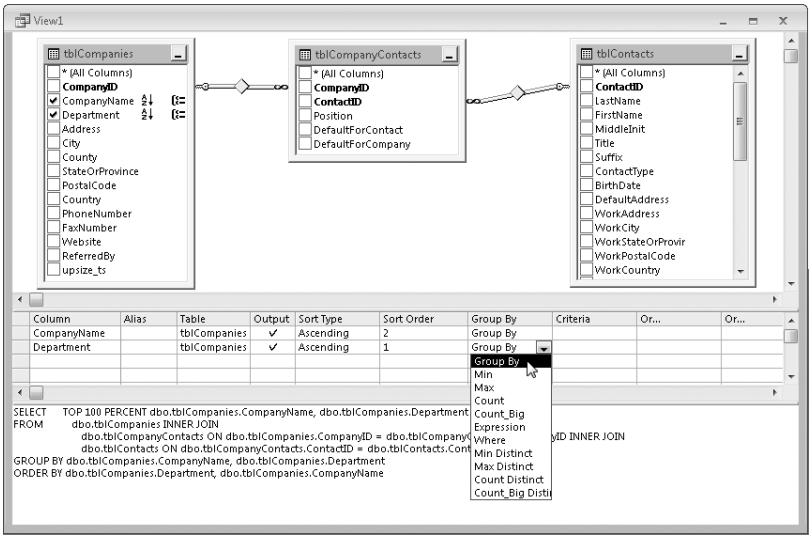

Figure 27-12 Click the Group By button on the Ribbon to add the Group By column to the grid pane of the query designer.

Table 27-1 lists the different options and functions that you can select in the query designer in the Group By column. Note that you can use some of these functions only on columns containing a numeric data type. As you can see, SQL Server supports many more functions than are available in a desktop database.

Table 27-1 SQL Server Query Designer Group By Options

Option	Description
Avg Avg Distinct	Returns the average of all non-null values in the column. Avg Distinct averages only the unique values. (Duplicates are ignored.) You can specify an average only on a numeric column.
Checksum_Agg Checksum_Agg Distinct	Returns the checksum value of an integer column, ignoring null values. Checksum_Agg Distinct calculates the checksum using only the unique values. (Duplicates are ignored.) You can specify checksum only on a column or expression that is the int data type.
Count Count Distinct Count_Big Count_Big Distinct	Returns the count of all non-null items in the column. Count_Big is the same as Count except it always returns a bigint data type, whereas Count always returns an int data type. Count Distinct and Count_Big Distinct count only unique values. (Duplicates are ignored.)
Expression	Indicates that the column contains an expression using one or more aggregate functions.
Group By	Groups the aggregate calculations on the values in this column. You can specify Group By for more than one column.
Max Max Distinct	Returns the highest value in the column (the last value alphabetically for text data types). Max Distinct, included for compatibility with the ANSI SQL-92 standard, considers only the unique values in the column but does not return a result different from Max. All null values are ignored.
Min Min Distinct	Returns the lowest value in the column (the first value alphabetically for text data types). Min Distinct, included for compatibility with the ANSI SQL-92 standard, considers only the unique values in the column but does not return a result different from Min. All null values are ignored.
StDev StDevP	Returns the standard deviation for all values in the column. StDevP returns the standard deviation for the population of all values in the column. The column can contain only numeric values.
Sum Sum Distinct	Returns the total of all values in the column. Sum Distinct totals only the unique values. (Duplicates are ignored.) The column can contain only numeric values.
Var VarP	Returns the statistical variance for all values in the column. VarP returns the statistical variance for the population of all values in the column.
Where	Specifies that this column or expression is not included in an aggregate expression in the SELECT list or in the GROUP BY clause, but is available to filter the rows before the query forms the groups. Access generates a WHERE clause immediately following the FROM clause in the SQL that defines your query.

A useful tool when you're building queries in the query designer is the Verify SQL Syntax button. You can find this button in the Tools group on the Design tab to the right of the Property Sheet button. When you click the button, Access calls SQL Server to verify

that the SQL syntax of your query is correct. If SQL Server finds any errors, Access returns a message stating the nature of the syntax error. This tool is especially helpful if you are writing SQL directly in the SQL pane or if you are evaluating an SQL statement from the text editor. (See "Building Queries Using a Text Editor" later in this chapter for more information.)

Viewing Other Properties

You can specify a variety of additional properties for a query, such as the number of rows you want returned, lookups for the columns returned, and information about any input parameters you might use. To view the properties for any query type in Design view, right-click anywhere in the diagram pane or grid pane and click Properties on the shortcut menu, or click the Property Sheet button in the Tools group on the Design tab. The properties for each query type are different because each query type supports different features. You will learn more about the properties for each type of query later in this chapter.

Working with Views

Views are the simplest type of query that SQL Server supports. Views offer a way for you to control what data is displayed and the order in which it is displayed. Views are great tools for the SQL Server developer because they allow the developer to control access to the data in the tables. Instead of assigning individual user permissions to certain parts of a table, you can create a view and give the user permissions to see and change data only through the view.

The following are some reasons you might use a view:

- You want a secure and updatable replacement for a table. With a properly designed view, your users can still add, delete, and update rows, but they don't need permission to access the underlying tables directly. You can give users permission to use the view, and if you want, you can restrict access to specific columns simply by removing them from the view.
- You want to organize the recordsets that you use for your forms and reports. You can use a view as the record source for any form or report or as the row source for any combo box or list box. If you carefully plan a naming convention, you can create one view for each form or report to make it easy to know which object uses the recordset. For example, vwFrmCompanies might be the view that supplies the recordset for the frmCompanies form.
- You want to add querying power to your recordsets. Unlike a table, views can automatically sort or filter information based on fixed date ranges or predetermined values. For example, if you have users who need to work only with the invoices for the current month, you can build a view as a recordset that displays invoices only for the current month.
- You can use views as a record source anywhere that you can use a table as a record source. You can also use functions and stored procedures as record sources in your views.

> **Note**
>
> Views are usually meant to provide updatable recordsets, but this isn't always the case. Views are not updatable when you include an aggregate function that summarizes the data or you group the data. Views are also partially updatable if you are building a recordset from multiple tables because only the many side of the recordset provided by a view is updatable. Keep this in mind as you construct your views.

Now that you know about some of the uses for views, finish building the view that you started building previously. To keep the first example fairly simple, start by removing some elements from the existing view. First, remove the tblCompanyContacts and tblContacts tables by right-clicking them and selecting Remove from the shortcut menu. Also, if you turned on the Group By option, turn it off by clicking the Group By button in the Output Operations group on the Design tab. When you are done, your view should look like Figure 27-13.

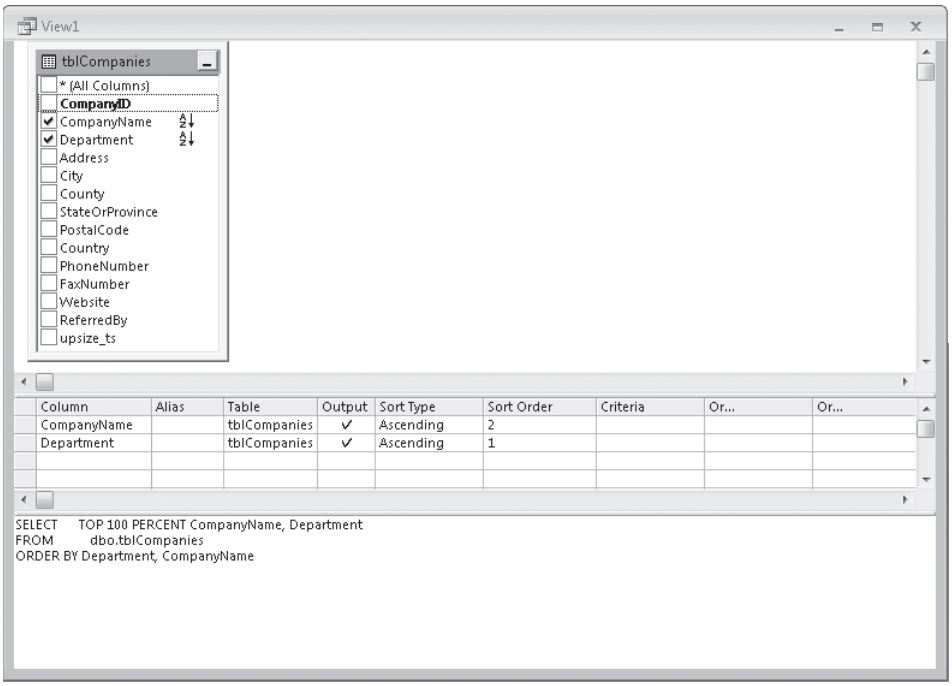

Figure 27-13 Here is a view in the query designer with only the Companies table included.

Design this view so that it would be a suitable record source for the Companies form. Include all the columns from the tblCompanies table in the result set by placing a check mark next to the name of each column in the diagram pane. You do not need to include the upsize_ts field because that is a non-updatable system field that Access uses to help make update commands more efficient. You can also select all columns by placing a

check mark next to the *(All Columns) option, but because you want to specify a sort on one of the output columns, add each of the columns individually. (There is a way to specify a sort when using the *(All Columns) option, as you will see in "Working with In-Line Functions" on page 1513.)

Because you previously added the CompanyName and Department fields to the view, Access adds the CompanyID field below the Department field when you select it. Click the selection button next to the CompanyID field in the grid pane, and drag it to the top of the list.

This view will be used as the record source for the Companies form, so it will be easier to browse through the records of the Companies form if they are sorted by company name. You also want the company name to be the only column that the result set is sorted on, so remove the Sort Type from the Department column by changing the value in the list to Unsorted. Notice that when you do so, Access automatically adjusts the sort order for the CompanyName column to 1. Click the Save button on the Quick Access Toolbar to save the view. In the Save As dialog box, type **vwCompanies** as the name of the view, and click OK. The completed view should look like Figure 27-14. You can find this view saved as vwXmplCompanies in the sample database.

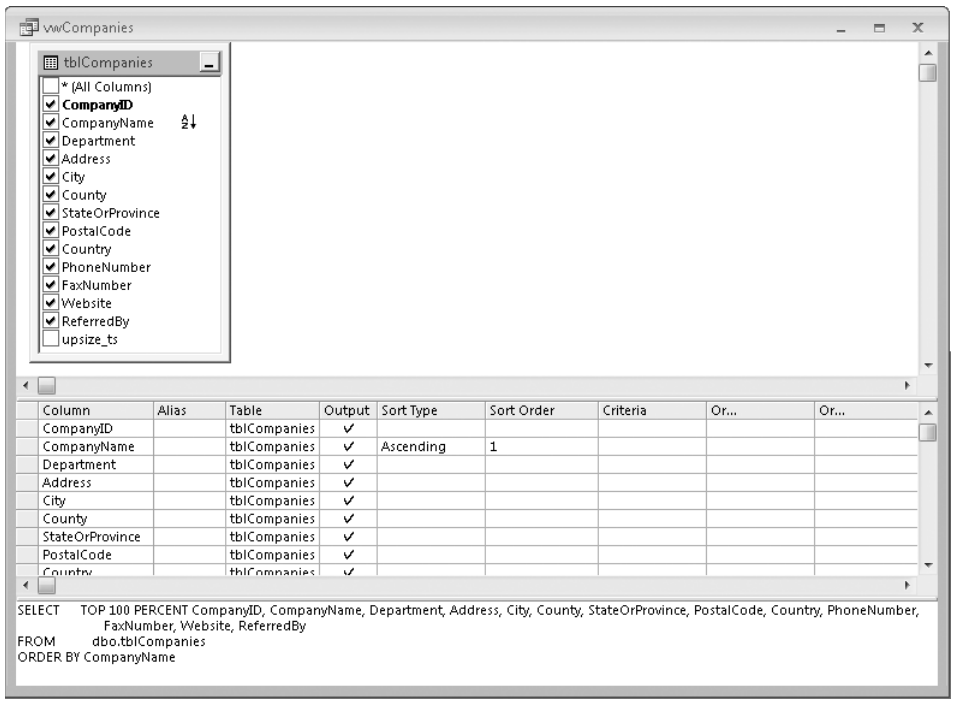

Figure 27-14 Your completed vwCompanies view should look like this.

> **Note**
>
> You'll notice throughout this chapter that you are instructed to save a query each time you modify it before you switch to Datasheet view. Unlike a query in a desktop database, an Access project must save your query in SQL Server before it can run it for you. If you attempt to switch to Datasheet view before you have saved your changes, Access displays a warning dialog box informing you that you must first save your changes. You can click Yes to save your changes and run the query or click No to return to Design view.

After you save the view, take a look at the result set by clicking the View button in the Views group on the Design tab. (Datasheet view is the default for this button when you have a view open in Design view.) Notice that all the columns for the tblCompanies table are displayed, but they are not sorted alphabetically by company name.

Although you can specify ORDER BY in the query designer, the columns appear not sorted because SQL Server 2005 honors the sort only to resolve the TOP clause. When returning all rows (the default TOP 100 PERCENT clause inserted by Access), the server doesn't bother to sort the rows. To run the query, Access sends an SQL statement of SELECT * FROM vwCompanies to the server. If you intend to use this view as a record source for a form or report or as a row source for a combo box or list box, you'll need to specify a SELECT statement on the view in the row source or record source and add the ORDER BY clause. For instance, you'll need to use an SQL statement like SELECT * FROM vwCompanies ORDER BY CompanyName. If your project file is connected to an SQL Server 2000 database, that version of the server does return the rows sorted without requiring the additional ORDER BY clause.

> **Note**
>
> You can update columns and delete or insert rows in your vwCompanies view. As with any Datasheet view in Access, you add new rows by entering data in the new row at the end of the datasheet, so new rows won't necessarily be in alphabetical order by company name. You will see the rows in the correct order the next time you open the view.

Before moving on to create more complex queries using in-line functions, take a look at the different properties you can specify for a view. Right-click anywhere in the design area and click Properties on the shortcut menu, or click the Property Sheet button in the Tools group on the Design tab. Figure 27-15 shows the Properties window for the vwCompanies view.

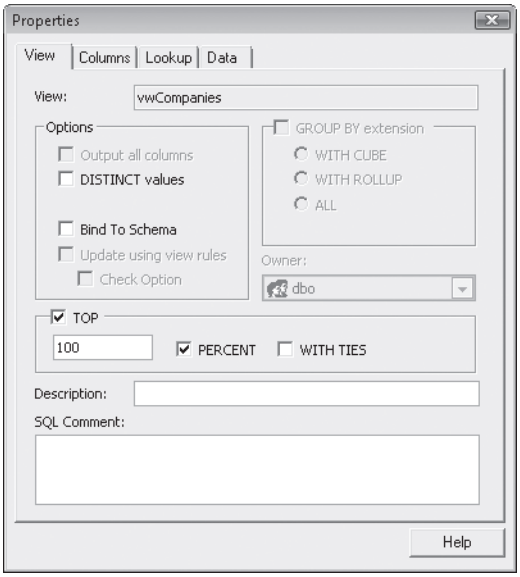

Figure 27-15 The Properties window for views displays four tabs where you can adjust various properties.

INSIDE OUT

SELECT TOP 100 PERCENT

You might have noticed that when you specify the Sort Type property for any column, Access automatically adds a TOP 100 PERCENT clause to the SQL statement for the view. This clause returns all (100 percent) of the records returned by the SELECT statement starting from the beginning (top) of the recordset. The clause might seem redundant, but it is required in order for the results to be sortable. In the current ANSI SQL standard, views return unordered recordsets, and the standard does not allow the ORDER BY clause in a view. The standard also does not define the TOP clause. SQL Server 2000 and later, however, define an extension to the standard that allows you to include an ORDER BY clause in a view, but only when you also include the nonstandard TOP clause. SQL Server also allows you to specify a TOP clause without also including an ORDER BY clause.

You might remember from Chapter 8 that queries in a desktop database also support a TOP clause and an ORDER BY clause; however, you can include an ORDER BY clause without also specifying TOP. If you think about it, using TOP without an ORDER BY specification doesn't make sense unless you want a specific number of arbitrary rows returned by the query. When you include TOP and ORDER BY, the query returns the top rows based on the sort sequence you specified.

You should remember the requirement in SQL Server to add the TOP clause to any SELECT statement that includes an ORDER BY clause, especially when working in the text editor to create views, functions, and stored procedures. Keep in mind, however, that when you use TOP 100 PERCENT, SQL Server 2005 does not return the rows sorted.

You can see four tabs in the Properties window for a view. The Columns, Lookup, and Data tabs contain properties similar to those that you can specify for a table. The Columns tab lets you override the extended properties for each individual column in the view such as Description, Caption, Format, and Decimal Places. The Lookup tab allows you to define lookups for each column. The Data tab allows you to specify filters and subdatasheet information. For more information on these properties, refer to "Understanding Column Properties" on page 1467. The tab you might not be familiar with is the View tab. Table 27-2 lists the properties that you can set on the View tab.

Table 27-2 Options on the Properties Window View Tab

Feature	Description
View	The currently assigned name for this view. You cannot change this value in the Properties window, but it is updated if you save the view under a different name.
Output All Columns	Selecting this check box is the same as selecting the *(All Columns) option in the diagram pane. All columns in the table(s) are returned.
DISTINCT Values	The view displays only distinct values and filters out all duplicate rows.
Bind To Schema	Selecting this check box prevents all users from modifying the design of any underlying record sources for this view (tables, functions, and stored procedures). This is a useful option because if anyone changes the underlying record sources, this view (and any other view, function, or stored procedure based on the same record sources) could become invalidated and no longer function properly.
Update Using View Rules	When you construct a view using multiple record sources and then you perform updates via the view, Access often sends update commands to the server using the base table that was changed, not the view. When you select this check box, Access always performs update commands using the view.
Check Option	If you select the Update Using View Rules option, you can select this check box to ensure that any row modified in the view also satisfies any conditions in the WHERE clause of the view. This prevents users from inserting or changing a row that would be eliminated by the criteria in the WHERE clause.
GROUP BY Extension	These additional options are available if you specified Group By in the view: WITH CUBE specifies that the view should return a multidimensional result set for the columns specified in the GROUP BY clause that summarizes all combinations of aggregate functions and columns in the view. WITH ROLLUP specifies that the view should return a multi-dimensional result set for the columns specified in the GROUP BY clause that provides a singular summary for each combination of aggregate functions and columns in the view. ALL includes all duplicates in the result set. (By default, GROUP BY eliminates duplicates.)

Feature	Description
Owner	The SQL Server owner of the view. The default owner is dbo (database owner).
TOP	Select this check box so that the view results can be sorted and only the top percentage or number of rows are returned. PERCENT specifies a percentage of rows to return instead of a number. WITH TIES returns all tied rows when more than one row qualifies for the cutoff of the TOP limit because they contain duplicate values on the sorting criteria. For example, if you request TOP 10 WITH TIES and the sorted column contains five rows with the value 1 and eight rows with the value 2, the view returns 13 rows because multiple rows contain the same value as the 10th row.
Description	Stores a description of the view in the view definition.
SQL Comment	Allows you to add a comment about the view. This comment can be seen by anyone who can view the SQL syntax for the view.

Working with In-Line Functions

When working with in-line functions in the query designer, you can think of them as a balance between the simplicity of views and the power of stored procedures. Like views, in-line functions have the ability to return recordsets that can be edited and updated. Like stored procedures, in-line functions can use parameters. Parameters are variables that are passed to the function that can control the way it behaves. For example, you can create a function that retrieves records that were created within a certain date range and use parameters to specify the beginning and end dates of the date range when the function is executed.

The following are some reasons you might use an in-line function:

- You need an updatable recordset that can be built based on conditional parameters. For example, you might need to edit the contact information for every contact at a certain company. You can build an in-line function that fetches the contacts only for the company you specify and edit their information directly in the query results.
- In-line functions are great for returning aggregate data, such as a count of records or an average retail price. Views can return this information as well, but the requirements for fetching this data often rely on user-provided information, which you can easily ask for using an in-line function with parameters.
- On a similar note, in-line functions are a good record source for reports. Reports are often produced using a specific date range or a particular person. In-line functions give you the ability to build the record source based on the specific needs of the person retrieving the report.

- You can use an in-line function as a record source anywhere that you can use a table or view as a record source. You can also use views, other functions, and stored procedures as record sources in your in-line functions.

INSIDE OUT

Construct In-Line Functions Carefully

When working with in-line functions, you will find it easier to create a function that returns a recordset that is not updatable. The same design considerations that cause view recordsets to be nonupdatable also apply to in-line functions. However, because in-line functions are often used to aggregate data, they will frequently return recordsets that can't be updated. Keep this difference in mind when deciding which type of query to build. If you need a recordset that is updatable but doesn't require parameters, use a view. If you need to aggregate data or accept parameters, use an in-line function. If you need to build a function that is updatable, pay close attention to how you build an in-line function and always test it to make sure that you can update the recordset it returns.

Now that you know more about some of the uses of in-line functions, let's build one in the query designer. For this example, you'll build a function that returns the contact information for every contact associated with a specific company. You'll use multiple joins to associate the contact information with the company information, and you'll sort the results by each contact's last name and first name. To select the rows, you'll use a parameter to specify the company for which you want to return the contact information.

Open the Contacts.adp sample database, and click the Query Wizard button in the Other group on the Create tab. Access opens the New Query dialog box, previously shown in Figure 27-2. Select Design In-Line Function in the New Query dialog box, and then click OK to start a new in-line function. From the Add Table dialog box, add the tblCompanies, tblCompanyContacts, and tblContacts tables, and then close the dialog box. Notice that when you add the tables to the diagram pane, Access automatically displays the three joins that represent the relationships between these tables. As you did before when creating a view using these tables, click the join line between the tblContacts table and the tblCompanies table, and press Delete to remove it. If you open the SQL pane, you can see that Access has also created the appropriate INNER JOIN clauses in the FROM clause. These are the exact joins that you need to use for this query to fetch related rows from the three tables. Next, specify which columns you want to return in the query. Select the CompanyName column from the tblCompanies table and the *(All Columns) option from the tblContacts table. When you are done, the query designer window should look like Figure 27-16 (with the SQL pane displayed).

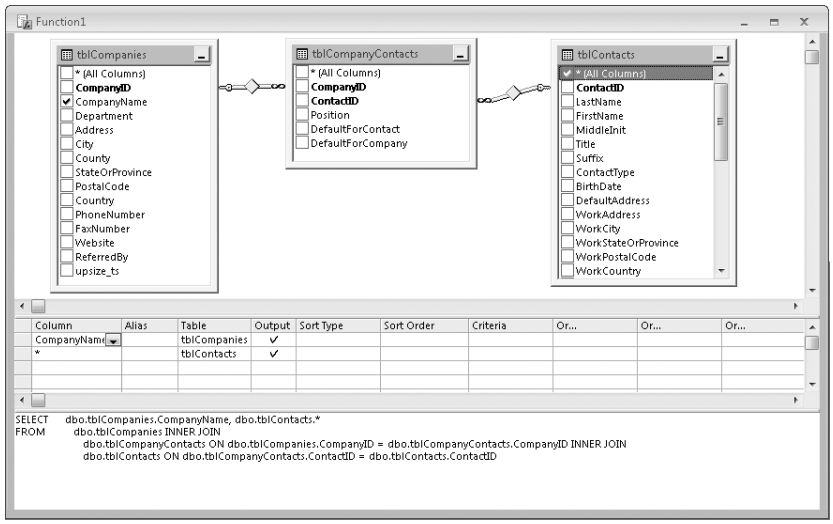

Figure 27-16 Your new in-line function in the query designer joins three tables.

Click the Save button on the Quick Access Toolbar, and name the function fctCompany-Contacts in the Save As dialog box. Next, click the View button in the Views group on the Design tab to view the in-line function results. Notice that all contact information is returned unsorted. You still have more work to do before this function will work the way you want.

Return to Design view so that you can make further modifications to the function. Now you need to specify the parameter that filters the company for which you want the contact information. As with queries in a desktop database, you can use a parameter anywhere that you could otherwise specify a literal. So, you can use a parameter in a criteria expression for the CompanyName column. In a query in an Access project, you declare a parameter by using the @ symbol followed by the name of your parameter. Unlike parameters in an Access desktop database, you cannot use blanks or other special characters in a parameter name, but you can use the underscore character to provide spacing in the name. To allow the user to enter all or part of a company name, in the criteria for the CompanyName column enter the following:

```
LIKE @Enter_Company_Name + N'%'
```

The capital letter *N* preceding the '%' indicates that you want a National Language literal to be used in the concatenation. A National Language (or Unicode) literal uses two bytes for each character and supports a more extensive range of characters. Because the CompanyName column is a Unicode field (data type nvarchar), if you type a simple string literal, Access inserts the capital letter *N* for you to ensure that the characters in your string can be compared with the full range of characters that might be stored in the column. Note that a filter icon now appears next to the CompanyName column in the diagram pane to indicate that you have defined a criterion for this column.

Now the function is complete. Click the Save button on the Quick Access Toolbar to save your changes. The finished function should look like Figure 27-17.

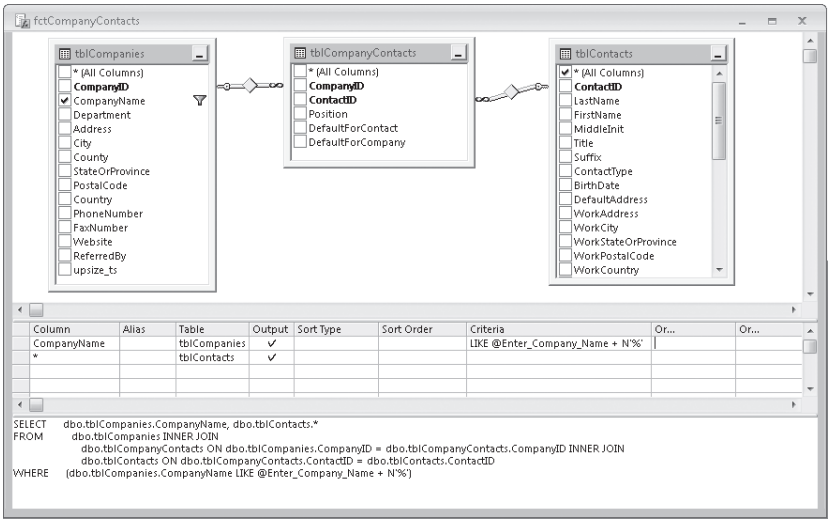

Figure 27-17 Your in-line function now includes a parameter used as a criterion to filter the CompanyName column.

> **Note**
>
> When you attempt to save the function for a second or subsequent time, Access might prompt you with an ADO error message saying that the Aggregate Type property cannot be added for each output field in the function. A bug in the initial release of Access 2007 displays this error. Click OK in each message, and Access will indeed save your changes back to the server.

Now you're ready to test your new function. Switch to Datasheet view, and Access displays an Enter Parameter Value dialog box, as shown in Figure 27-18.

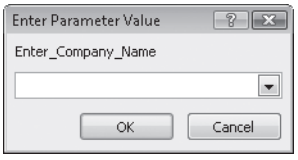

Figure 27-18 Access displays the Enter Parameter Value dialog box when you run a query that uses parameters.

The sample database contains several company names that begin with the letter *C*, so type that letter in response to the parameter prompt, and click OK. Access displays the result, as shown in Figure 27-19.

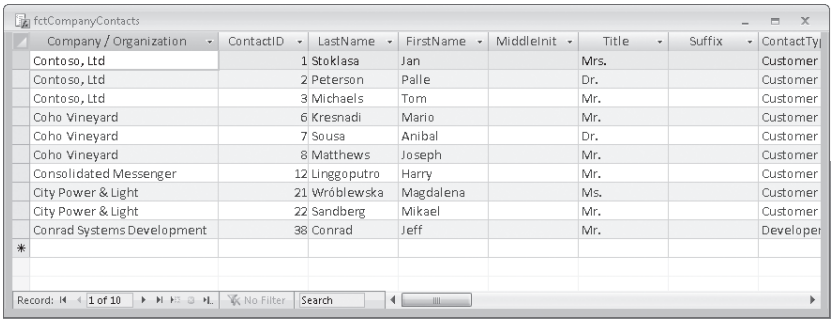

Figure 27-19 You should see these results when searching for contacts associated with companies that have a name beginning with the letter ***C***.

Notice that the query found five different companies having names that begin with the letter *C*. You might note that the results are not sorted in any way. You could add an ORDER BY clause to this query to sort the results, but, as we previously mentioned, SQL Server 2005 does not return the results sorted when you use an ORDER BY clause—the same problem we pointed out when working with views.

INSIDE OUT

Use Descriptive Parameter Names

It is always a good idea to use descriptive names for the parameters you use in functions and stored procedures, even if you plan to hide them from the user with code or a form interface. First, a descriptive name helps you remember how the query uses the parameter. Second, if a user runs the query from the Navigation Pane, the name of the parameter should explain what you want the user to enter.

Before moving on to stored procedures, take a quick look at the different properties you can specify for a function. Return to Design view, right-click anywhere in the design area, and click Properties on the shortcut menu. Access displays the Properties window for the in-line function, as shown in Figure 27-20.

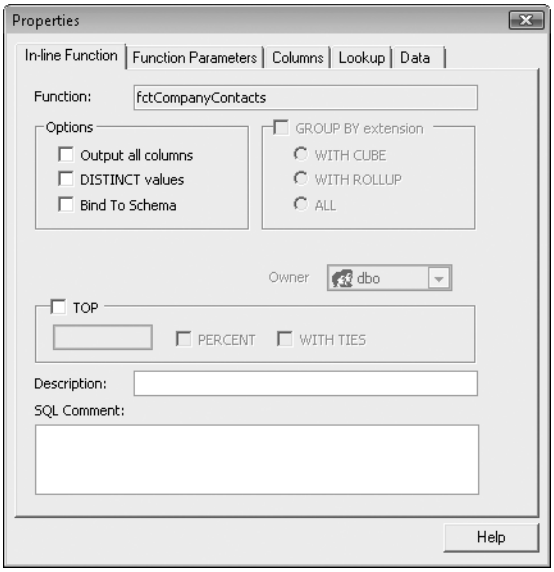

Figure 27-20 The Properties window for an in-line function includes an In-line Function tab.

The tabs available in the Properties window for in-line functions are similar to those for views, except an in-line function includes an In-Line Function tab and a Function Parameters tab. The Columns, Lookup, and Data tabs are the same as the corresponding tabs in the Properties window for views. The In-Line Function tab offers many of the same options as the View tab in the Properties window for views. However, an in-line function does not offer the Update Using View Rules or Check Option options. For more information on the other options on the In-Line Function tab, refer back to Table 27-2 on page 1512.

Click the Function Parameters tab to view the list of parameters specified for this in-line function, as shown in Figure 27-21. This is a good way to verify all the parameters you specified for an in-line function (or a stored procedure). You can't add parameters in this window because Access wouldn't know where the parameters should be used in the query. However, you can specify a data type and a default value for parameters that you have already created. Keep in mind that you're entering a literal value in the Default column, so you must enclose character literals and date/time literals in single quotes.

The figure shows that we've entered a default value of N'Viescas'. You can find this query saved as fctXmplCompanyContacts in the sample database. When you run this query, you can choose <DEFAULT> in the drop-down list in the Enter Parameter Value dialog box, and the query will return the records for companies that have a name beginning with that character string. Close the query when you are finished.

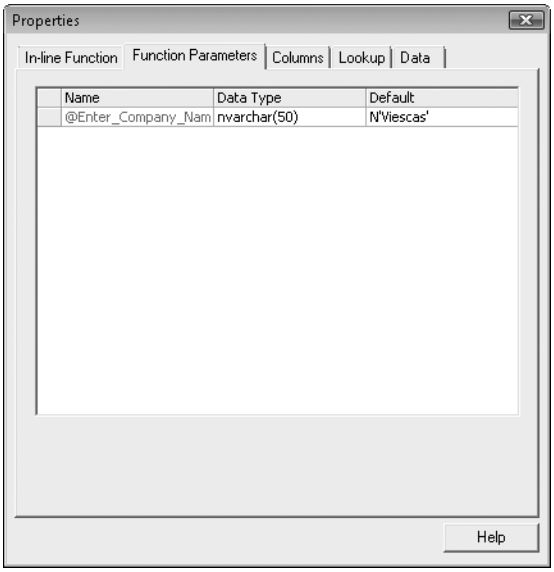

Figure 27-21 The Properties window for an in-line function includes a Function Parameters tab.

Working with Stored Procedures

Stored procedures are the most powerful type of query you can build in the query designer. These queries have most of the capabilities of views and in-line functions, plus the added ability to perform action queries (updating, deleting, and adding data). The real power and potential of stored procedures is considerably beyond what the query designer is capable of building. As the name implies, stored procedures allow you to write complex procedures that include logic testing (If...Else), looping (While), creating temporary tables, and error trapping using Transact-SQL. However, you will find plenty of situations where building a stored procedure in the query designer comes in handy.

The primary use of stored procedures that you build in the query designer is to perform action queries. Like functions, stored procedures can accept parameters and can return updatable recordsets. So, you can use a stored procedure as the record source of a form or report. However, you cannot use stored procedures as the record source of other views or in-line functions.

Because the best use for a stored procedure built in the query designer is to perform an action query, let's build a stored procedure that inserts (*appends* in Access terminology) a record into the tblCompanies table. To perform this task, you have two options. You can use an append query (an INSERT statement that uses a query to fetch one or more rows to be inserted), or you can use an append values query (an INSERT statement that uses a VALUES clause to specify the column values for one new row).

Both methods work fine for adding a record to the tblCompanies table, but the append values syntax is more straightforward, especially when you need to insert only one row.

It also has the added benefit of being easier to convert to a text stored procedure, which you will do later in this chapter.

Begin by opening the Contact.adp project, and then click the Stored Procedure button in the Other group on the Create tab. Alternatively, you can click the Query Wizard button in the Other group, select Design Stored Procedure in the New Query dialog box, and then click OK. Add the tblCompanies table to the new stored procedure from the Add Table dialog box, and then close the dialog box. Be sure the SQL pane is visible by clicking the SQL button in the Tools group on the Design tab under Stored Procedure Tools so that you can watch Access build this query in SQL. This will be helpful information to know when you build one yourself later on in the text editor.

Next, specify that this is an append values query. To do this, click the Append Values Query button in the Query Type group on the Design tab, as shown in Figure 27-22. The query designer window changes and shows only two options in the grid pane: Column and New Value.

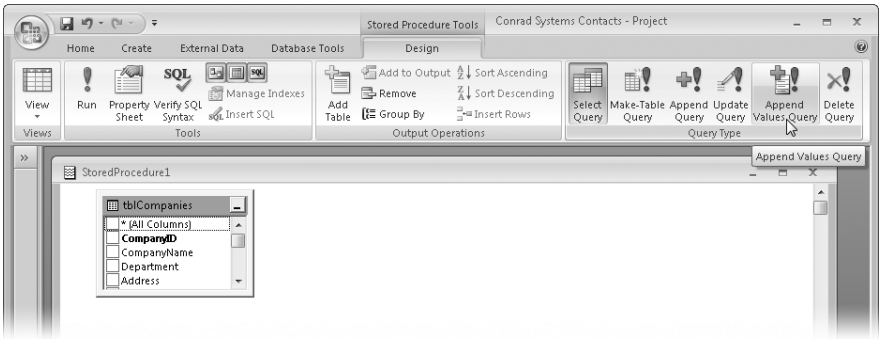

Figure 27-22 Click the Append Values Query button in the Query Type group to modify your stored procedure to insert the values you specify.

Use Column to specify which columns you want to append to the Companies table, and use New Value to indicate the values you want to add for each column. Notice that you can select which columns you want to append by clicking the box to the left of each column name in the diagram pane. When you select a column, Access displays a plus sign (rather than a check mark) in the box. Because you want to add an entire row to the tblCompanies table, go ahead and click next to each column name except the CompanyID and upsize_ts columns. (The CompanyID column is the AutoNumber data type, so SQL Server automatically generates the next value when you insert a row, and the upsize_ts column is a system-maintained timestamp value that helps Access improve the performance of updates.) When you are done, the query designer should look like Figure 27-23.

Now you need to specify the values that you want to append to the tblCompanies table. You could simply enter values in the New Value field for each column and execute the stored procedure. However, the next time you wanted to add a row to the Companies table, you would have to build the procedure all over again.

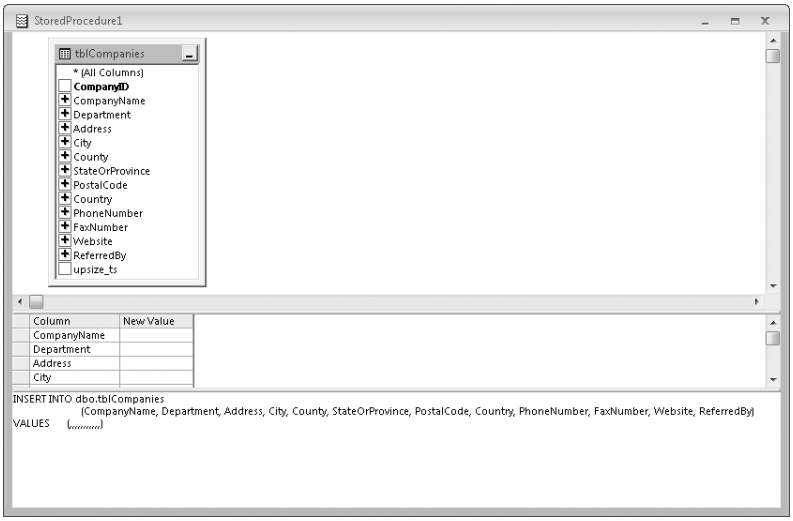

Figure 27-23 Select the columns to which the stored procedure will append values.

Instead, you can use parameters for each value. When you do this, Access prompts you for the values you want to insert into a new row each time you run the stored procedure. You can also execute the stored procedure from Visual Basic code and supply the column values by setting the parameters. To make each new value a parameter, you need to enter a valid parameter name next to each column name. Remember, a parameter name must begin with the @ character and cannot contain blanks or special characters other than the underscore character. Refer to the following list for suggested parameter names for each column name:

Column Name	Parameter Name
CompanyName	@CompanyName
Department	@Department
Address	@Address
City	@City
County	@County
StateOrProvince	@StateOrProvince
PostalCode	@PostalCode
Country	@Country
PhoneNumber	@PhoneNumber
FaxNumber	@FaxNumber
Website	CAST(@Website AS nvarchar(50))
ReferredBy	@ReferredBy

Notice that you must convert (CAST) the @Website parameter to an nvarchar data type because the data type of the Website column is ntext and the query designer doesn't accept a parameter for an ntext data type column. Converting the parameter to an nvarchar data type allows the parameter to be used to append data to the Website column. Interestingly, when you enter this parameter, Access displays a dialog box telling you that the conversion might be unnecessary. However, if you attempt to enter a simple parameter name, Access tells you that this produces a data type conversion error and won't let you leave the New Value box until you correct it.

After you enter all the parameter names, the stored procedure is complete. Click the Save button on the Quick Access Toolbar, and name the procedure spAddOneCompany. Your finished stored procedure should look like Figure 27-24. You can also find this query saved as spXmplAddOneCompany in the sample database.

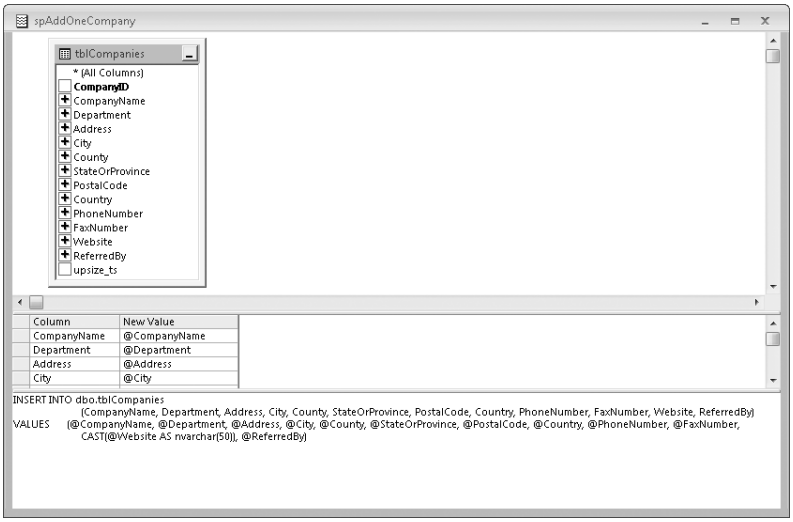

Figure 27-24 Your completed spAddOneCompany stored procedure should look like this.

You can see the stored procedure in action by clicking the arrow on the View button in the Views group on the Design tab and clicking Datasheet view or by clicking the Run button in the Tools group on the Design tab. Access prompts you for each of the parameter values in succession. Input a value for each parameter, and click OK, but only the CompanyName field is required to successfully save a new row. (The ReferredBy parameter is an integer value that identifies the ContactID of the person who referred this company. Enter a valid ContactID, or leave the value blank.) When the procedure is complete, you receive a message stating that the procedure completed successfully but didn't return any records. Now open the tblCompanies table in Datasheet view. Notice that the information you provided appears as a new row in the table.

CAUTION

When working with action queries in an Access desktop database (.accdb), you can switch to Datasheet view and see a recordset that represents the potential changes to the data. The desktop database action query doesn't actually execute until you click the Run button. This is not the case with stored procedures in the query designer in a project file (.adp). Viewing a stored procedure is the same as executing it. Be careful how you build your action queries, and always test them on a backup copy of your tables before executing them against your production tables.

Before moving on to text queries, let's take a look at the properties for a stored procedure. Because the stored procedure you've been building is an append query, you won't be able to see all the available properties for a stored procedure in the Properties window for your query. Open the sample spXmplCompanyParameter query in Design view—this is a simple stored procedure on the tblCompanies table that uses a parameter to filter the CompanyName column. As before, you can view the properties for a stored procedure by right-clicking anywhere in the design area and clicking Properties on the shortcut menu. Access displays the Properties window for the stored procedure, as shown in Figure 27-25.

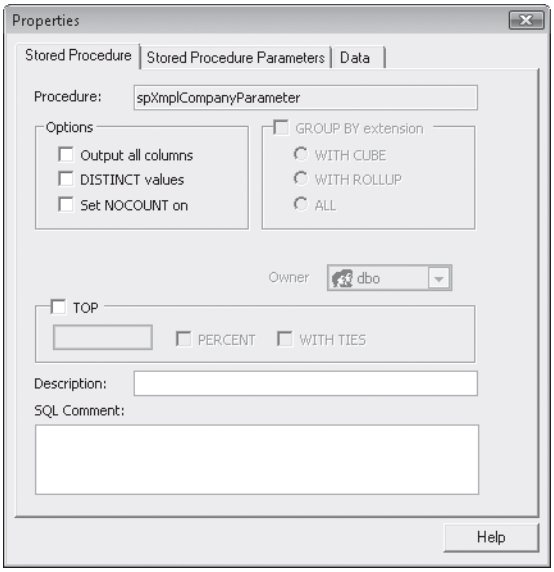

Figure 27-25 Open the Properties window of a stored procedure to see the Stored Procedure tab.

Notice that, unlike a function or view, you cannot define extended properties for columns (such as Caption or Format properties), and you cannot define a lookup field for any output column. You can, however, define filters and subdatasheet information on the Data tab just like you can for views and functions. The information and options

on the Stored Procedure Parameters tab are identical to those that you can find on the Function Parameters tab for a function, as you saw earlier in Figure 27-21.

On the Stored Procedure tab, you can see many options that are similar to those available for views and functions. The one option unique to stored procedures is Set NOCOUNT On. Normally when you execute a stored procedure, the procedure returns a count of the rows affected. When you execute the procedure from another procedure, you can query the number of rows affected by referencing the built-in @@ROWCOUNT function. When you select the Set NOCOUNT On check box, the stored procedure won't return the record count. For some complex procedures that you use in an Access project, you should select this check box. When the procedure includes multiple SELECT statements and one or more statements early in the procedure return a row count of 0, Access assumes that the output from the procedure will be zero rows. If the final SELECT returns a recordset, Access won't display it if NOCOUNT is set to off (the default). By setting NOCOUNT to on, the procedure doesn't return any row count, so Access displays the final record set correctly. Close the Properties window and the sample spXmplCompanyParameter query.

For more information on the remaining properties, refer back to Table 27-1 on page 1506.

Building Queries Using a Text Editor

Thus far you have learned how to build queries using the query designer. You have already discovered that you can duplicate the functionality of Access desktop database (.accdb) queries by using the similar interface provided by the Access project query designer. However, SQL Server offers substantially more querying power than can be harnessed by using only the graphical query designer. To get the most out of SQL Server's querying power, you must create queries using Transact-SQL in the text editor.

You won't find any commands to build a query in the text editor on the Ribbon. To begin building a query using the text editor, you can click the Query Wizard button in the Other group on the Create tab. When you do this, Access displays the New Query dialog box, as shown in Figure 27-26.

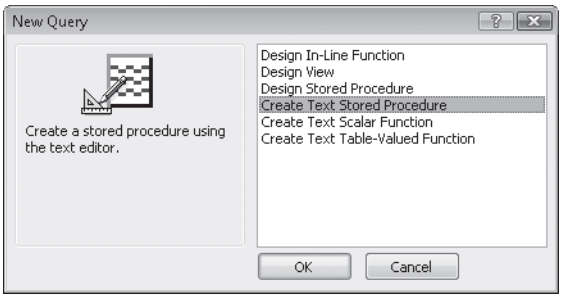

Figure 27-26 The New Query dialog box in an Access project file presents options to build queries using a text editor.

You can see that this dialog box gives you options to begin designing in-line functions, views, and stored procedures in the query designer, which we've previously discussed. The remaining three types of queries are the ones that you can build in the text editor—text stored procedures, text scalar functions, and text table-valued functions.

> **Note**
>
> SQL Server actually supports only three types of queries: views, functions, and stored procedures. To make designing the common select and action queries easier, Access projects offer the query designer versions of these query types. Because functions and stored procedures can be so much more robust than the query designer can visually display, Access projects also offer the option to design these query types using a text editor to take better advantage of their available features.

To create a text query, you must use Transact-SQL. Transact-SQL (or T-SQL) is a special form of SQL used by SQL Server that includes extra commands and features to extend its capabilities. Think of Transact-SQL as "Visual Basic for SQL Server." It extends the capabilities of SQL by introducing control-of-flow statements (such as IF...ELSE and WHILE), parameter inspection during execution, error trapping, complex mathematical calculations, and transaction batching. Transact-SQL is robust enough to perform many of the actions for which you would normally use Visual Basic or some other programming language. But the real beauty of Transact-SQL is that the queries are compiled and run from SQL Server, thus increasing the speed and performance of your database.

Because Transact-SQL is such a powerful and complex language, this chapter only scratches the surface when describing its features. The next section, "Building a Text Stored Procedure," shows you step by step how to build a fairly complex stored procedure in the Contacts database using Transact-SQL. The remaining two sections provide brief descriptions of how to use text scalar functions and text table-valued functions along with examples to get you started building some of your own.

The text editor itself is a relatively simple tool that offers some features that can make the job of writing text-based queries a little easier. When you open a new query in the text editor, some of the syntax is already provided for you. The provided syntax lays the groundwork for the type of query you want to create and also includes some of the default commands that you might need to use in most text queries of that type. In fact, the provided syntax is the only difference between creating one type of query (such as a stored procedure) over another (such as a table-valued function). If you want, you can change one query type into another (as long as you haven't yet saved the query) simply by changing the provided syntax in the text editor.

If you want to add SQL statements to the editor, you can type them or you can use the query designer to build one. To use the query designer to build an SQL statement, you can right-click in the text editor and click Insert SQL on the shortcut menu, or you can

click the Insert SQL button in the Tools group on the Design tab. When you are done creating the query in the query designer (which Access opens as a dialog box), close the query designer. Access asks you whether you want to insert the SQL you created into the text editor. If you click Yes, Access copies the SQL you created to the text editor window at the location where you last placed the cursor.

You can also use the query designer to edit an existing SQL statement. To do this, highlight the SQL statement, right-click the highlighted selection, and click Edit SQL on the shortcut menu. You can also click the Edit SQL button in the Tools group on the Design tab. (Note that the Insert SQL button changes to an Edit SQL button in the Tools group when you highlight the SQL statement.) Either method displays the highlighted text in a query designer dialog box where you can modify it or check the SQL syntax using the syntax checker.

CAUTION!

Save your work often, and keep backup copies of your text queries. (Notepad works great for this purpose.) The text editor does not have any undo features and can be very unforgiving if you make a mistake.

Building a Text Stored Procedure

The most powerful query type that you can create in an Access project is the stored procedure. Thus far, you've had a glimpse of what stored procedures can do by designing one using the query designer. To take full advantage of the capabilities of stored procedures, you must create them using the text editor. Here are some of the abilities that you have with a text stored procedure:

- Use just about any SQL statement including SELECT, DELETE, UPDATE, and INSERT.
- Modify or create other database objects. You can create tables and views as well as modify their properties and contents.
- Execute other stored procedures and functions. You can use any available stored procedure, function, or system variable by calling it or executing it from a stored procedure.
- Declare parameters and variables. You can input and output specific data as well as declare variables that are referenced and changed within the stored procedure.
- Use control-of-flow statements. Much like the statements available in Visual Basic procedures, control-of-flow statements in stored procedures allow you to check for conditions, loop through statements multiple times, and trap errors.
- Group multiple statements using transactions. You can make sure an entire group of operations completes successfully by batching them inside a transaction and monitoring any errors. If any operation fails, you can ask SQL Server to roll back all changes made to the database within the transaction.

Let's take the stored procedure you created in the query designer and rebuild it in the text editor. Along the way, you'll improve it by declaring variables, adding control-of-flow statements, including error trapping, and calling another stored procedure.

Beginning a New Stored Procedure in the Text Editor

Open the Contacts.adp sample project file. Click the Query Wizard button in the Other group on the Create tab, and select Create Text Stored Procedure in the New Query dialog box (Figure 27-26). Click OK, and Access opens the text editor with some Transact-SQL code and comments already inserted, as shown in Figure 27-27.

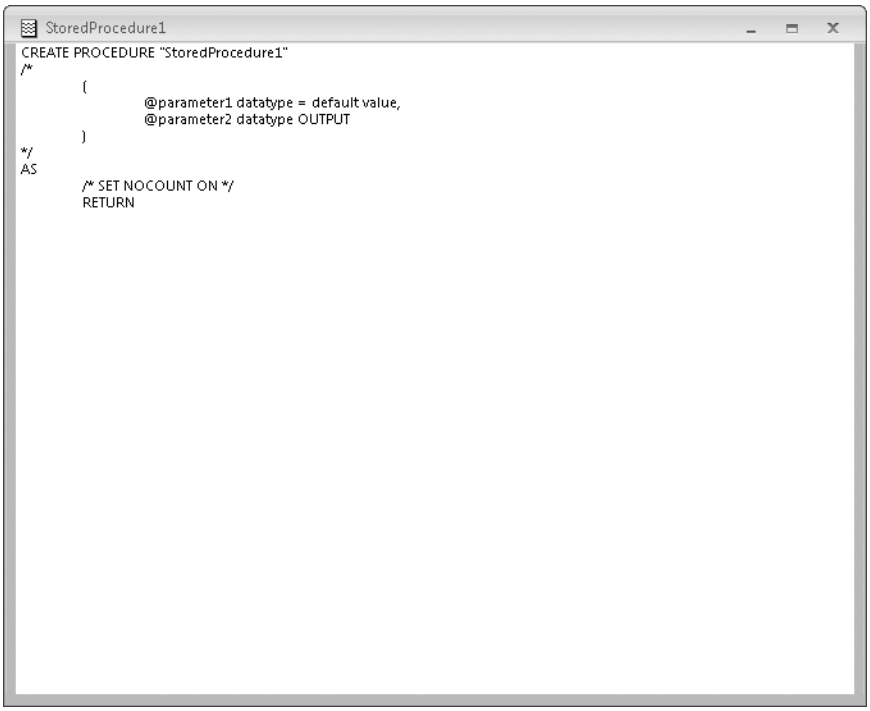

Figure 27-27 Here you can see a new stored procedure in the text editor.

You can create a stored procedure using any text-editing tool and import it into the query builder of SQL Server Management Studio. In order for SQL Server to recognize that you are trying to create a stored procedure, you need to use the CREATE PROCEDURE statement followed by the name of the procedure in double quotes. Similarly, if you want to change an existing stored procedure, you use the ALTER PROCEDURE statement followed by the name of the procedure you want to alter. (You don't need to use quotes when referring to an existing procedure unless the procedure name contains blanks or special characters.) If you were to save this procedure, close it, and reopen it in Access, you would see the ALTER statement instead of the CREATE statement, because you would be altering an existing procedure.

The /* and */ symbols mark the beginning and end of a block of comments. Anything written within those symbols is designated as a comment and is there only to provide information about the purpose of the stored procedure. You can also enter a single-line comment by typing two hyphens (--) at its beginning. Here are a couple of examples:

```
/* This is a multiple-line block of text that describes
a section of the stored procedure */

-- This is a single-line comment
```

> **Note**
>
> You can see that the only keywords in the skeleton stored procedure shown in Figure 27-27 that aren't within comment blocks are CREATE, AS, and RETURN. Access includes the parameter declaration section and the SET NOCOUNT ON statement encased in block comments because it assumes that you won't always want to use these sections. However, the syntax of these sections is correct, so it's a simple matter to remove the comment symbols to make them a part of your procedure.

If your procedure requires parameters, you must declare them enclosed in parentheses immediately following the procedure name. Declare parameters for each item of data that you need to input into the procedure or that you will output from the procedure. To declare a parameter, enter the name of the parameter (the first character must be @), one or more spaces, and the data type of the parameter. When the data type is one of the text data types, you can also specify the maximum length in parentheses following the data type name. For some numeric data types, you can optionally specify the precision and scale in parentheses following the data type name. See Table 26-1 on page 1461 for a list of data types supported by SQL Server.

You can optionally specify a default value by following the data type specification with an equal sign and the value that you want to assign as the default. Remember to enclose character and date/time literals in single quotes. By default, all parameters are input parameters. If you want to use a parameter for output, use the OUTPUT keyword at the end of the parameter declaration. If you want to declare multiple parameters, separate them with commas. You will learn more about parameters as you work with them in the example in this section. Here is an example that creates a single input parameter called @City that is the data type nvarchar, has a size of (30), and has a default value of Null:

```
(
@City nvarchar(30) = NULL
)
```

By default, SQL Server returns a row count message (the number of rows returned/affected) to Access after every SQL statement it executes. This is a useful feature if you are monitoring the row count for each SQL statement internally in the procedure. Remember, as we discussed earlier, if you are using multiple SQL statements in your

procedure and the first one returns no rows, Access assumes that the entire procedure doesn't return any rows. If you execute a statement afterward that *does* return rows (in Datasheet view for example), Access does not display them. Recall also that using the SET NOCOUNT statement followed by the ON keyword solves this problem. Because this statement is already included in the skeleton stored procedure, simply remove the beginning and end comment markers to enable it.

You enter the body of your stored procedure after the SET NOCOUNT statement and before the RETURN statement. This includes all SQL statements, control-of-flow statements, and output information. When you re-create the spAddOneCompany stored procedure, you will place the INSERT statement here.

The RETURN statement tells the stored procedure to exit the procedure unconditionally. If you are evaluating conditions in the middle of the procedure and need to stop execution for any reason, you can use the RETURN statement to do so.

Re-Creating the Stored Procedure from the Query Designer Example

Based on the information covered so far, you might have already guessed what you need to do to re-create the spAddOneCompany stored procedure that you built previously in the query designer. You must declare an input parameter for each value you need to add to the tblCompanies table. Then add the INSERT statement after the AS statement to add the record to the table. In this example, you don't need SET NOCOUNT ON because the procedure contains only one SQL statement. The code to accomplish this is as follows:


```
CREATE PROCEDURE sptAddOneCompany
-- This procedure creates a new record in the Companies table
        (
         @Company_Name nvarchar(50),
         @Department nvarchar(50) = Null,
         @Address nvarchar(255) = Null,
         @City nvarchar(50) = Null,
         @County nvarchar(50) = Null,
         @State_Or_Province nvarchar(20) = Null,
         @PostalCode nvarchar(20) = Null,
         @Country nvarchar(50) = N'United States',
         @Phone_Number nvarchar(30) = Null,
         @Fax_Number nvarchar(30) = Null,
         @Website ntext = Null,
         @Referred_By int = Null
                                  )
AS
INSERT INTO dbo.tblCompanies
     (CompanyName, Department, Address, City, County, StateOrProvince,
     PostalCode, Country, PhoneNumber, FaxNumber, Website, ReferredBy)
VALUES
     (@Company_Name, @Department, @Address, @City, @County,
     @State_Or_Province, @PostalCode, @Country, @Phone_Number, @Fax_Number,
     @Website, @Referred_By)
RETURN
```

INSIDE OUT

Working in the SQL Designer

You can easily build this code in the skeleton procedure by inserting a blank line after the AS keyword and then starting the SQL designer by clicking the Insert SQL button in the Tools group on the Design tab. Follow the steps from earlier in this chapter to create the INSERT statement that you need, and then close the designer. Click Yes when Access offers to paste the SQL into your procedure. You can also open the spAddOneCompany procedure that you created earlier (or the sample spXmplAddOneCompany query), copy the SQL to the Clipboard, and paste it into your new procedure.

You can find this procedure saved as sptXmplAddOneCompany in the sample database. Click Save on the Quick Access Toolbar, and save the query as sptAddOneCompany. When you execute this stored procedure, you are prompted for each input parameter. You can enter a value for each input, or select <DEFAULT> or <NULL> from the drop-down list. (In most cases, the default is null.) Be sure to enter a value for the company name. When you are done, you will receive a message as shown in Figure 27-28.

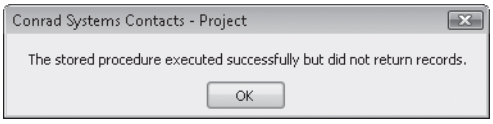

Figure 27-28 If you entered a company name, you should see this message stating the stored procedure executed successfully.

There were no records returned because you didn't specify any in the stored procedure. If you open the tblCompanies table, you will see that the record was added successfully. If you don't enter a company name, you receive a cryptic message like the one shown in Figure 27-29 because the company name is a required column in the Companies table, and you won't find a new row in the tblCompanies table.

Figure 27-29 You'll see this message stating the stored procedure failed if you did not include a company name when prompted.

As the stored procedure is written, there is no way to prevent this message from occurring if the user doesn't provide a company name. However, you could use several options to control the input in the stored procedure. You could use a form with ADO code in a Visual Basic module that controls the input, requires the user to enter a company name, and then passes it to the stored procedure. You could also modify the stored procedure itself to check for the correct values and return more descriptive information if it fails.

> **Note**
>
> If you remove the RETURN statement in the example text stored procedure, save it, and open it again in Design view, Access displays it using the query designer because you haven't added any SQL commands that cannot be represented in the designer. As long as the stored procedure is only a single SQL statement with no other clauses, Access prefers to show it to you in the designer. The rest of the examples in this chapter can be edited only by using the text editor.

Adding Control-of-Flow Statements to Your Stored Procedure

Before you modify the stored procedure, you need to learn about some new statements you can use to check the company name. In this section, you will learn how to

- Declare variables, assign values to them, and test them
- Include IF/ELSE statements that test variables and execute an alternate set of statements
- Include BEGIN/END statements to define a block of statements that SQL Server should execute after an IF or ELSE statement
- Use the @@ROWCOUNT and @@IDENTITY system variables

You have already learned how to create and use parameters in your procedures. You can also declare variables that store data that you can work with internally while the procedure is executing. Declaring variables is similar to declaring parameters, except you must use the DECLARE statement, which must follow the AS statement. You can declare multiple variables in one DECLARE statement, separated by commas. For example, the following code creates two variables called @CompanyID and @retMsg:

```
DECLARE @CompanyID int,
        @retMsg varchar(150)
```

The default value for all variables is Null. If you want to assign a value to a variable, you can use either the SELECT statement or the special SET statement. Both statements have similar syntaxes. The following are examples:

```
-- Assign a value to the @retMsg variable using SELECT
SELECT @retMsg = 'This message value was created using the SELECT statement.'
--Assign a value to the @retMsg variable using SET
SET @retMsg = 'This message value was created using the SET statement.'
```

With the SELECT statement, you can assign values to multiple variables by separating the assignments with commas. For example, the following SELECT statement assigns values to two variables:

```
SELECT @retMsg = 'Assign two values.', @CompanyID = 0
```

With the SET statement you can assign only one value per statement. After you have declared a variable, you can use it anywhere in your procedure that you would normally be able to use a value of the same data type.

IF/ELSE statements allow you to test a condition and execute alternate tasks. When you use an IF statement, you can include any valid expression that can evaluate to true or false. You can also use a SELECT statement that evaluates to true or false, but you must encase it in parentheses. If SQL Server evaluates the IF statement as true, it executes the next line of code following the IF statement. If SQL Server evaluates the IF statement to false, it looks for an ELSE statement. If there is an ELSE statement, SQL Server executes the code immediately following the ELSE statement. If there is no ELSE statement, SQL Server continues executing the rest of the procedure after the IF statement. An example of an IF statement with an ELSE statement follows:

```
IF @Company_Name is Null
RETURN

ELSE
SET @retMsg = 'The company name was provided.'
```


When the condition in the IF statement evaluates as true, SQL Server executes only the next statement immediately following the IF statement. So what do you do if you want to execute multiple statements based on the evaluation of a single IF statement? One option is to include the IF statement again, with a new line of code after the second and subsequent IF statements. Another option is to create a batch of code using the BEGIN/END statements. SQL Server treats all lines of code between a BEGIN statement and an END statement as a single batch or line of code. If you want to execute multiple statements based on an IF condition, use the BEGIN and END statements to batch them. For example:

```
If @Company_Name is Null
  BEGIN
    SET @retMsg = 'You must supply a company name.'
    SELECT @retMsg AS "Status"
    RETURN
  END
```

In this example, if the @Company_Name variable is Null, then the procedure assigns an informative message to a character variable, returns that variable in a column named Status, and exits using a RETURN statement. Notice the use of the AS statement. This keyword specifies that the value of @retMsg should be returned as a recordset with the column heading *Status*. If the IF statement evaluates to true, you see a result similar to Figure 27-30. You can find an example procedure that includes this test saved in the sample database as sptXmplAddOneCompanyTest.

SQL Server provides a variety of default system stored procedures and variables that you can use to improve the performance of your procedures. One useful variable that SQL Server provides is the @@ ROWCOUNT variable. (The names of system variables always begin with two @ symbols.) @@ ROWCOUNT stores the number of rows that were returned by the last SQL statement that was executed, useful for determining

whether any records were returned. Another useful variable is the @@IDENTITY variable. After executing an SQL statement that updates, adds, or deletes a row that contains an identity column (equivalent to an AutoNumber in a desktop database), the @@ IDENTITY variable contains the identity value of the last updated row.

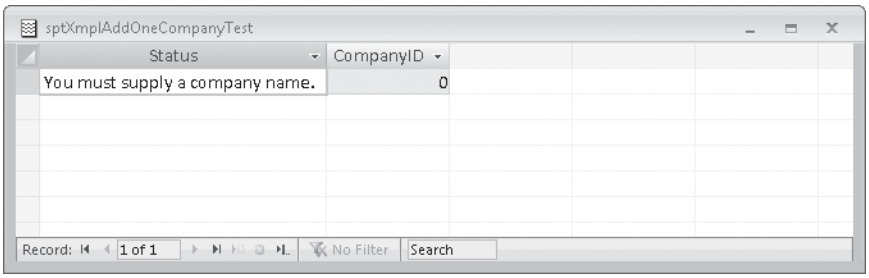

Figure 27-30 This example shows a procedure returning an error message as a recordset when you do not supply a company name.

INSIDE OUT

Download Resources for Help on These Topics

To learn more about the system variables, functions, and stored procedures that are available to improve the performance of your own user-defined stored procedures, refer to Microsoft *SQL Server 2005 Books Online*. Remember that you can download a free copy of the Books Online (a set of help files) from *www.microsoft.com/technet/prodtechnol/sql/2005/downloads/books.mspx*.

Now that you've learned about some more of the available features in a stored procedure, you can solve the problem that occurs if a user fails to supply a company name. As you probably guessed, you need to use an IF/ELSE statement to evaluate the input of the @Company_Name parameter. Modify your sptAddOneCompany procedure to look like the following:

```
ALTER PROCEDURE sptAddOneCompany
-- This procedure creates a new record in the Companies table
        (
         @Company_Name nvarchar(50),
         @Department nvarchar(50) = Null,
         @Address nvarchar(255) = Null,
         @City nvarchar(50) = Null,
         @County nvarchar(50) = Null,
         @State_Or_Province nvarchar(20) = Null,
         @PostalCode nvarchar(20) = Null,
         @Country nvarchar(50) = N'United States',
         @Phone_Number nvarchar(30) = Null,
         @Fax_Number nvarchar(30) = Null,
         @Website ntext = Null,
```

```
        @Referred_By int = Null
                                  )
AS

SET NOCOUNT ON

-- Declare a variable to return the new company ID
--  and a status message
DECLARE @CompanyID int,
        @retMsg varchar(150)

-- Make sure there's a company name
If @Company_Name is Null
  BEGIN
    -- Nope.  Set and return an error message and exit.
    Set @CompanyID = 0
    SET @retMsg = 'You must supply a company name.'
    SELECT @retMsg AS "Status", @CompanyID AS "CompanyID"
    RETURN
  END

-- Company name supplied, so attempt the insert
INSERT INTO dbo.tblCompanies
     (CompanyName, Department, Address, City, County, StateOrProvince,
     PostalCode, Country, PhoneNumber, FaxNumber, Website, ReferredBy)
VALUES
     (@Company_Name, @Department, @Address, @City, @County,
     @State_Or_Province, @PostalCode, @Country, @Phone_Number, @Fax_Number,
     @Website, @Referred_By)

-- If rowcount is not zero
If @@ROWCOUNT > 0
  -- Row inserted successfully
  BEGIN
    SET @CompanyID = @@IDENTITY
    SET @retMsg = 'New Company Added Successfully'
  END

ELSE
  -- Insert failed, return a zero company ID and an error message
  BEGIN
    SET @CompanyID = 0
    SET @retMsg = 'There was an Error. No Company was added.'
  END

-- Return the status of the insert and the new ID, if any
SELECT @retMsg AS "Status", @CompanyID AS "CompanyID"
RETURN
```

Not only does this improved procedure check the value of the @Company_Name parameter, but it also returns a recordset at the end of the procedure to let you know whether it succeeded. To do this, it uses another set of IF/ELSE statements to evaluate

the @@ROWCOUNT variable and assign values to the @retMsg and @CompanyID variables. When the insert is successful, the procedure uses the @@IDENTITY variable to find out the ID of the new company row so that it can return the value as part of the successful insert message. The SELECT statement at the end of the procedure returns the message and the company ID. You can find this query saved in the sample database as sptXmplAddOneCompanyTest.

Grouping Multiple Statements with Transactions

One of the great features of stored procedures and Transact-SQL is that you can control multiple SQL statements in one stored procedure. So far you've learned how to add a new record to the tblCompanies table and check for required values before executing the INSERT command. What if the business rules for this database specifically state that you cannot add a company without adding at least one valid contact? You could create two separate procedures—one to add a company and another to add the contact—but that won't guarantee that the rows in each table are related. Creating separate procedures also means that your users might execute one procedure and not the other and the job ends up only half complete.

The solution is to use an SQL Server transaction. When you use a transaction, you can batch all your SQL statements into one procedure and commit the changes to the tables only if all the SQL statements succeed. If any errors occur or any constraints are violated, you can roll back the entire transaction and avoid having only a part of your procedure succeed. Creating a transaction in a procedure involves using three Transact-SQL statements: BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION. Using these statements is fairly simple. You indicate the beginning of a transaction batch by using the BEGIN TRANSACTION keywords. Then you include all the SQL statements that you want to be a part of the transaction. If all the statements succeed, you use the COMMIT TRANSACTION statement to commit the changes to the database. If your conditional statements within the transaction identify that an error or a failure occurred, you use the ROLLBACK TRANSACTION statement to undo all the changes made by any SQL statements in the transaction batch.

The basic syntax for a transaction batch is simple. The difficult part is using effective conditional statements to evaluate whether there is a problem that requires you to roll back the transaction. One particularly useful system variable for doing this is the @@Error variable. You can check the @@Error variable after the execution of each SQL statement. If @@Error is nonzero, you know that you must roll back the transaction.

You can use a transaction to execute multiple SQL statements in a single procedure, but that doesn't mean that all the SQL statements should be a part of the procedure. Each procedure should be straightforward and have a unique purpose. When you create the stored procedure that adds the company row and the contact row, you will create a separate procedure to add the contact. This way, you will have one procedure with the singular purpose of creating a complete company and another with the purpose of adding a contact.

After you have created a procedure that adds the contact information to the contacts table, you can execute it from your sptAddOneCompany procedure. In order for the

contact procedure to tell the company procedure that it succeeded, you need to use output parameters that it can pass back to the company procedure. Let's take a look at the procedure that adds the contact information (sptAddContact):

```
CREATE PROCEDURE sptAddContact
-- This procedure creates a new record in the Contacts table
     (
          @Last_Name nvarchar(50),
          @First_Name nvarchar(50) = Null,
          @Middle_Init nvarchar(1) = Null,
          @Title nvarchar(10) = Null,
          @Suffix nvarchar(10) = Null,
          @Contact_Type nvarchar(50) = Null,
          @Birth_Date datetime = Null,
          @Default_Address smallint = 1,
          @Work_Address nvarchar(255) = Null,
          @Work_City nvarchar(50) = Null,
          @Work_State_Or_Province nvarchar(20) = Null,
          @Work_Postal_Code nvarchar(20) = Null,
          @Work_Country nvarchar(50) = 'United States',
          @Work_Phone nvarchar(30) = Null,
          @Work_Extension nvarchar(20) = Null,
          @Work_Fax_Number nvarchar(30) = Null,
          @Home_Address nvarchar(255) = Null,
          @Home_City nvarchar(20) = Null,
          @Home_State_Or_Province nvarchar(20) = Null,
          @Home_Postal_Code nvarchar(20) = Null,
          @Home_Country nvarchar(50) = 'United States',
          @Home_Phone nvarchar(30) = Null,
          @Mobile_Phone nvarchar(30) = Null,
          @Email_Name ntext = Null,
          @Website ntext = Null,
          @Photo image = Null,
          @Spouse_Name nvarchar(75) = Null,
          @Spouse_Birth_Date datetime = Null,
          @Notes ntext = Null,
          @Commission_Percent float = 0,
          @ContactID int OUTPUT
                              )

AS
SET NOCOUNT ON

-- Exit procedure if @Last_Name is null
IF @Last_Name IS NULL
BEGIN
     SET @ContactID = 0
     RETURN
END

-- Otherwise, perform the insert
      INSERT INTO tblContacts
```

```
          (LastName, FirstName, MiddleInit, Title, Suffix, ContactType,
          BirthDate, DefaultAddress, WorkAddress, WorkCity,
          WorkStateOrProvince, WorkPostalCode, WorkCountry, WorkPhone,
          WorkExtension, WorkFaxNumber, HomeAddress, HomeCity,
          HomeStateOrProvince, HomePostalCode, HomeCountry, HomePhone,
          MobilePhone, EmailName, Website, Photo, SpouseName, SpouseBirthDate,
          Notes, CommissionPercent)

     VALUES
          (@Last_Name, @First_Name, @Middle_Init, @Title, @Suffix,
          @Contact_Type, @Birth_Date, @Default_Address, @Work_Address,
          @Work_City, @Work_State_Or_Province, @Work_Postal_Code,
          @Work_Country, @Work_Phone, @Work_Extension,
          @Work_Fax_Number, @Home_Address, @Home_City,
          @Home_State_Or_Province, @Home_Postal_Code, @Home_Country,
          @Home_Phone, @Mobile_Phone, @Email_Name, @Website, @Photo,
          @Spouse_Name, @Spouse_Birth_Date, @Notes, @Commission_Percent)

If @@ROWCOUNT > 0
     SET @ContactID = @@IDENTITY
ELSE
     SET @ContactID = 0

RETURN
```

If you like, you can open the example sptXmplAddContact procedure in Design view, click the Microsoft Office Button, and then click Save As to save a new copy with a new name. As you can see, the sptAddContact procedure is fully capable of adding a complete contact record to the tblContacts table. It also includes @ContactID as an output parameter. If sptAddContact is called from any other procedure, it returns the @ContactID to let the calling procedure know whether the insert succeeded. If @ContactID is 0, no record was added. Otherwise, @ContactID is equal to the identity value of the newly added row.

Now that you've built the sptAddContact stored procedure, you're ready to update the sptAddOneCompany stored procedure. (You might want to open that query in Design view, click the Microsoft Office Button, click Save As, save a copy as sptAddCompanyAndContact, and then modify the new copy.) Not only do you need to populate the tblContacts and tblCompanies tables, you also need to add a record to the tblCompanyContacts linking table to create the relationship between the rows. As you might have guessed, you'll use a transaction to monitor these three INSERT actions and make sure they all complete successfully.

In order to execute the sptAddContact stored procedure, you need to use the EXECUTE command. To do so, you type the command followed by the name of the procedure you want to execute and any required parameters that you need to supply. You can provide the parameters in the order they are specified in the called procedure, or you can identify them specifically by name. The following is an example that identifies them specifically by name.

```
EXECUTE sptAddContact
          @Last_Name = @Contact_Last_Name,
          @ContactID = @intContactID OUTPUT
```

You must declare both @Contact_Last_Name and @intContactID as local variables. Adding the OUTPUT keyword to the @ContactID assignment in the procedure call indicates that you expect the called procedure to return a value @intContactID. After the called procedure completes, you can use the local variable (@intContactID) to evaluate what was returned from the procedure—for example, checking it to make sure a ContactID was returned.

Now that you have learned about the various features and abilities of a stored procedure, it's time to put together a more complex procedure that uses everything covered so far. Your procedure after you have updated it to use transactions to monitor multiple SQL statements should be as follows:


```
ALTER PROCEDURE sptAddCompanyAndContact
/* This procedure creates a new record in the Companies table.
It also adds a corresponding record to the Contacts table and
completes the relation by adding a linking record to the
CompanyContacts table */
     (
          @Company_Name nvarchar(50),
          @Contact_Last_Name nvarchar(50),
          @Contact_First_Name nvarchar(50) = Null,
          @Contact_Extension nvarchar(20) = Null,
          @Department nvarchar(50) = Null,
          @Address nvarchar(255) = Null,
          @City nvarchar(50) = Null,
          @County nvarchar(50) = Null,
          @State_Or_Province nvarchar(20) = Null,
          @Postal_Code nvarchar(20) = Null,
          @Country nvarchar(50) = 'United States',
          @Phone_Number nvarchar(30) = Null,
          @Fax_Number nvarchar(30) = Null,
          @Website ntext = Null,
          @Referred_By int = Null,
   --Identifies whether the procedure succeeded:
          @retSuccess bit = 1 OUTPUT,
          @retMsg varchar(150) = Null OUTPUT
     )

AS
SET NOCOUNT ON

     DECLARE
          @intCompanyID int,
          @intContactID int,
          @retErr int          --Will capture any error messages

--Make sure all the variables and output parameters are set to the defaults
     SELECT @intCompanyID = 0, @intContactID = 0, @retSuccess = 1, @retErr = 0
```

```
-- Exit procedure if @Company_Name is null
IF @Company_Name IS NULL
BEGIN
     SELECT @intCompanyID = 0, @retMsg = 'You must supply a company name. '
     SELECT @intCompanyID AS 'New CompanyID', @intContactID AS 'New ContactID',
          @retMsg AS 'Last Status Message'
     RETURN
END

-- Exit procedure if @Contact_Last_Name is null
IF @Contact_Last_Name IS NULL
BEGIN
  SELECT @intContactID = 0, @retMsg = 'You must supply a contact last name. '
  SELECT @intCompanyID AS 'New CompanyID', @intContactID AS 'New ContactID',
         @retMsg AS 'Last Status Message'
     RETURN
END

/* We have values for @Company_Name and @Contact_Last_Name
    So let's begin the TRANSACTION and add the rows */
BEGIN TRANSACTION

-- First, add the company row
INSERT INTO dbo.tblCompanies
     (CompanyName, Department, Address, City, County,
     StateOrProvince, PostalCode,
     Country, PhoneNumber, FaxNumber, Website, ReferredBy)

VALUES
     (@Company_Name, @Department, @Address, @City, @County,
     @State_Or_Province, @Postal_Code,
     @Country, @Phone_Number, @Fax_Number, @Website, @Referred_By)

SET @retErr = @@Error  --Check for any errors
IF @retErr <> 0
     SELECT @retSuccess = 0, @intCompanyID = 0,
         @retMsg = 'Insert failed. No company was added. '

ELSE
     SELECT @retSuccess = 1, @intCompanyID = @@IDENTITY,
          @retMsg = 'New company added successfully '

--Next, add the contact row by Executing sptAddContact
--But don't do it if Company insert already failed
IF @retSuccess = 1
BEGIN
       EXECUTE sptXmplAddContact
            @Last_Name = @Contact_Last_Name,
            @First_Name = @Contact_First_Name,
            @Work_Address = @Address,
            @Work_City = @City,
            @Work_State_Or_Province = @State_Or_Province,
```

```
            @Work_Postal_Code = @Postal_Code,
            @Work_Country = @Country,
            @Work_Phone = @Phone_Number,
            @Work_Extension = @Contact_Extension,
            @Work_Fax_Number = @Fax_Number,
            @Website = @Website,
            @ContactID = @intContactID OUTPUT

       SET @retErr = @@Error  --Check for any errors
       -- Also test @intContactID returned by the procedure
       IF @retErr <> 0 Or @intContactID = 0
            SELECT @retSuccess = 0,
                 @retMsg = 'Insert failed. No contact was added. '
       ELSE
            SELECT @retSuccess = 1, @intContactID = @@IDENTITY,
                 @retMsg = 'New contact added successfully '
END

--Finally, add the row to the CompanyContacts table
--But don't do it if either previous insert failed
IF @retSuccess = 1
BEGIN
    INSERT INTO dbo.tblCompanyContacts
         (CompanyID, ContactID, DefaultForContact, DefaultForCompany)
    VALUES
        (@intCompanyID, @intContactID,1,1)


    SET @retErr = @@Error  --Check for any errors
    IF @retErr <> 0
         SELECT @retSuccess = 0,
           @retMsg = 'Insert failed. No company-contact was added.'
    ELSE
         SELECT @retMsg = 'New company-contact added successfully.'
END

If @retSuccess = 1
BEGIN
     COMMIT TRANSACTION
END

ELSE
BEGIN
     ROLLBACK TRANSACTION
END

-- Return the new company ID, contact ID, and status message.
-- The IDs will be zero if the procedure failed.
SELECT @intCompanyID AS 'New CompanyID', @intContactID AS 'New ContactID',
          @retMsg AS 'Last Status Message'

RETURN
```

The procedure uses a local bit variable, @retSuccess, to track the success or failure of the INSERT statements. When any statement fails, the procedure does not attempt to execute subsequent INSERT statements. At the end of the procedure, the code checks the value of @retSuccess and commits all updates if the value is true (1) or rolls back all updates if the value is false (0).

Also note that the company address and phone information is used for the work address and phone information for the contact. Because this is a company contact that you're creating, you can assume that they will be the same. You can leave out the contact's home address and other information that is not relevant to creating a useful company contact record.

You can find this procedure saved as sptXmplAddCompanyAndContact in the sample database. To test this procedure, you can open the tblContacts table in Design view and clear the Allow Nulls property for the FirstName column. Run the procedure and supply a company name and contact last name, but do not supply a contact first name. You should see the procedure fail, and you won't find a new company record saved in the database even though the INSERT to the tblCompanies table succeeded. Close this query.

Note

When you run the sptXmplAddCompanyAndContact stored procedure, Access prompts you for the two output parameters, @retSuccess and @retMsg. You should not enter values for these two parameters. You would normally execute this stored procedure from another query or a Visual Basic procedure. Because these two output parameters have default values declared, you do not need to supply these parameters when you execute this procedure from another procedure.

So far, we have covered many commands that are commonly used in stored procedures, but we have only scratched the surface. If you want to learn more about creating stored procedures and Transact-SQL in general, *SQL Server 2005 Books Online* is an excellent place to start.

Building a Text Scalar Function

Another type of query that you can build in the text editor is a text scalar function. Scalar functions are very similar in concept to the system functions that SQL Server provides. They are usually defined to serve a singular purpose and return a single value. For example, you can build a scalar function if you repeatedly need to compute a single aggregate value when working with recordsets in your database. You can build the scalar function and then execute it whenever you need to compute the value. For example, if you often need to know the average price of products sold in a given time range, you

can build a scalar function to compute that value for you. Then you only have to execute the scalar function instead of computing the value in the query every time you needed to use it.

Like stored procedures, scalar functions can use the many features of Transact-SQL. They can accept parameters, execute multiple SQL statements, and use variables. The key point to keep in mind with scalar functions is that they always return a single value.

Because scalar functions always return a single value, they can be used just about anywhere you can use a single value, variable, or parameter in other views, functions, and stored procedures. The flexibility of scalar functions makes them very useful as you design and implement the other queries in your database.

Note

When you refer to scalar functions or any other type of functions that you have built, you must always include the parentheses, (), after the name of the function in your EXECUTE statement. This is the standard syntax for calling a function and must be used even if the function does not accept any parameters.

Take a look at the syntax for creating a scalar function in the text editor. To begin, create a new scalar function by selecting Create Text Scalar Function from the New Query dialog box (Figure 27-26). Access creates a new scalar function, as shown in Figure 27-31.

The first half of the function should look familiar to you. The section in parentheses is where you declare any parameters that you will use in the scalar function. Remember to separate the parameters you declare with commas, and remember to remove the comment symbols (/* and */). Also note that you cannot declare OUTPUT parameters because scalar functions have only one output: the return value.

The RETURNS statement allows you to identify the data type of the value the scalar function is returning. The body of the scalar function must be entirely contained within one BEGIN/END statement block that follows the AS keyword. If you need to declare any local variables, define them within the BEGIN/END statement block. The last statement you use before the END keyword is the RETURN statement followed by the parameter or variable that contains the information the scalar function will return.

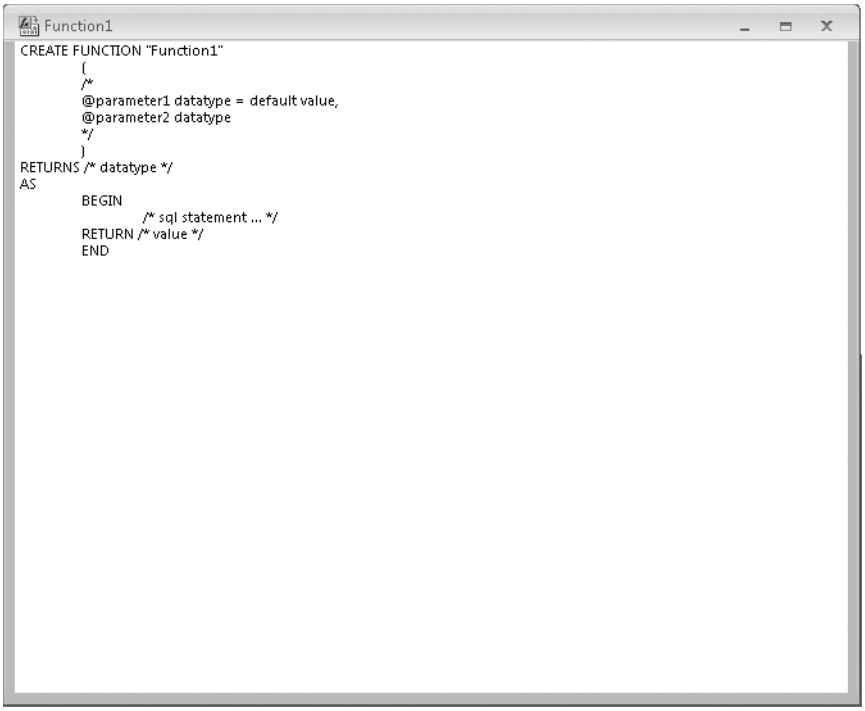

Figure 27-31 Click Create Text Scalar Function in the New Query dialog box to open a new scalar function in the text editor.

Building a Text Table-Valued Function

The last type of query that you can build using the text editor is a text table-valued function. In execution, the table-valued function is similar to a select query. It returns an entire recordset (or table) when it executes. The major difference is that table-valued functions can contain multiple SQL statements that are used to produce the final recordset that they return. As a result, table-valued functions can return tables that are the result set of several complex queries. The recordsets that table-valued functions return are read-only.

In a table-valued function, you can use the same Transact-SQL syntax as with scalar functions. The difference is a table-valued function returns a logical table instead of a single value, and it uses one or more INSERT statements to create the logical table that it returns. If you need to build a recordset that is based on multiple queries on other tables that might require parameters to determine the outcome, a table-valued function is very handy because it can perform this entire process in one function. Because table-valued functions return recordsets, you can use them anywhere you can use a single SQL statement in other views, functions, and stored procedures. Just remember that you can't alter the result set of a table-valued function.

Take a look at the syntax for creating a table-valued function in the text editor. You can begin to create a new table-valued function by selecting Create Text Table-Valued Function from the New Query dialog box (Figure 27-26). Access creates a new table-valued function, as shown in Figure 27-32.

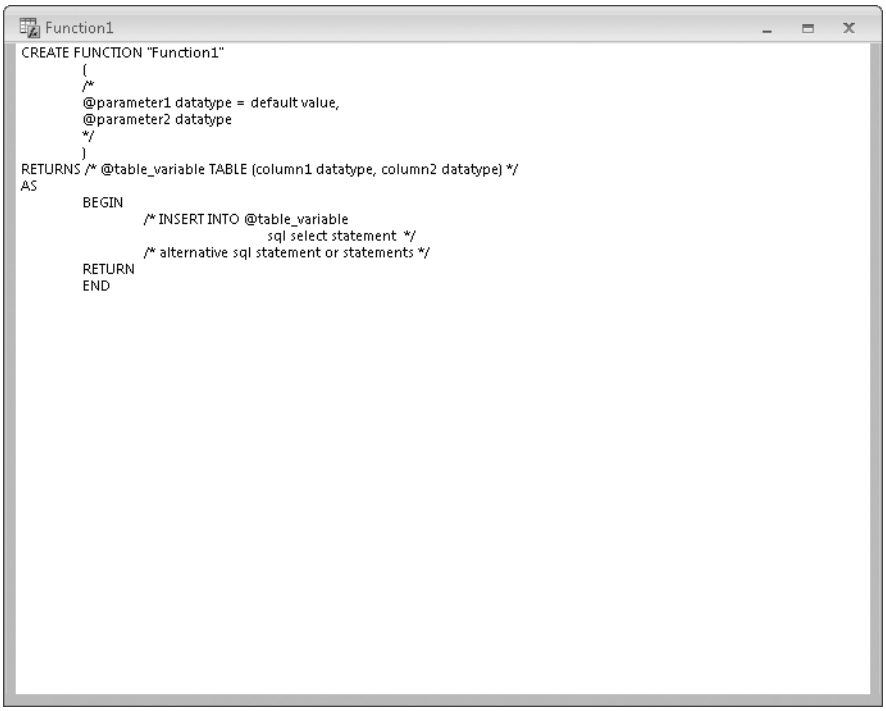

Figure 27-32 Click Create Text Table-Valued Function in the New Query dialog box to open a new table-valued function in the text editor.

The syntax for a table-valued function is very similar to the syntax for a scalar function. The first section in parentheses is where you declare any parameters you might need. Again, you can only declare input parameters because the only output of a table-valued function is a single table recordset. The RETURNS statement identifies the value that the function will return (in this case, a table variable). You must also specify the column names and data types for the table variable in the parentheses following the table variable name. Be sure to separate column declarations with commas. Of course, you should also remove the comment symbols (/* and */) that surround the required table declaration.

Similar to scalar functions, table-valued functions must contain the full body of the function within a BEGIN/END statement block that follows the AS keyword. You can use any number of SQL statements within the statement block, and you can also perform inserts, updates, and deletes on your declared table variable. If you need to use any local variables, be sure to declare them within the block. The last statement in the

BEGIN/END block must be a RETURN statement. SQL Server returns all rows that your code has inserted into the RETURNS table.

In this chapter you learned about the different types of queries that you can build in an Access project. You can see that in a project you have more query options available than you do in an Access desktop database (.accdb) and that the queries also offer powerful features that aren't available in a desktop database. Even though we took only a brief look at each type of query, you should have a good idea of the different features of each query type and how to best use them to accomplish your query tasks in an Access project. Combined with the knowledge you gained in the previous query-related chapters, you should be ready to tackle creating views, functions, and stored procedures of your own. The next chapter will cover creating forms in an Access project.