Stored Procedures

If views raise the bar of database functionality, then stored procedures take it to the next level. Unlike views, stored procedures can be used for much more than reading data. They provide a wide range of programming functionality. Categorically, stored procedures can be used to do the following:

- Implement parameterized views
- Return scalar values
- Maintain records
- Process business logic

Stored Procedures as Parameterized Views

As with views, stored procedures can be used to return a result set based on a `SELECT` statement. However, I want to clarify an important point about the difference between views and stored procedures. A view is used in a `SELECT` statement as if it were a table. A stored procedure is executed, rather than selected from. For most programming APIs, this makes little difference. If a programmer needs to return a set of rows to an application or report, any data access layer such as ODBC, OLEDB, ADO.NET, or SQL Client can be used to obtain results from a table, a view, or a stored procedure.

A stored procedure can be used in place of a view to return a set of rows from one or more tables. Earlier in this chapter, I used a simple view to return selected columns from the Product table. Again, the script looks like this:

```
CREATE VIEW vProductCosts
AS
SELECT ProductID, ProductSubcategoryID, Name, ProductNumber, StandardCost
FROM Production.Product
```

Contrast this with the script to create a similar stored procedure:

```
CREATE PROCEDURE spProductCosts
AS
SELECT ProductID, ProductSubcategoryID, Name, ProductNumber, StandardCost
FROM Production.Product
```

To execute the new stored procedure, the name is preceded by the `EXECUTE` statement:

```
EXECUTE spProductCosts
```

Although this is considered the most proper syntax, the `EXECUTE` command can be shortened to `EXEC`:

```
EXEC spProductCosts
```

In addition, when working with Management Studio, a stored procedure can be executed just by referencing its name:

```
spProductCosts
```

However, omitting the EXEC or EXECUTE command will not work when the stored procedure is referenced in other programming objects or programming interfaces. This abbreviated syntax is useful only in ad-hoc executions within Management Studio.

Using Parameters

A parameter is a special type of variable used to pass values into an expression. Named parameters are used for passing values into and out of stored procedures and user-defined functions. Parameters are most typically used to input, or pass values into, a procedure, but they can also be used to return values.

Parameters are declared immediately after the procedure definition and before the term AS. Parameters are declared with a specific data type and are used as variables in the body of a SQL statement. I will modify this procedure with an input parameter to pass the value of the ProductSubcategoryID. This will be used to filter the results of the query. This example shows the script for creating the procedure. If the procedure already exists, the CREATE statement may be replaced with the ALTER statement:

```
ALTER PROCEDURE spProductCosts
@SubCategoryID int
AS
SELECT ProductID, Name, ProductNumber, StandardCost
FROM Production.Product
WHERE ProductSubcategoryID = @SubCategoryID
```

To execute the procedure and pass the parameter value in SQL Query Analyzer or the Query Editor, simply append the parameter value to the end of the statement, like this:

```
EXECUTE spProductCosts 1
```

Alternatively, the stored procedure can be executed with the parameter and assigned value like this:

```
EXECUTE spProductCosts @SubCategoryID = 1
```

Stored procedures can accept multiple parameters and the parameters can be passed in either by position or by value, similar to the previous example. Suppose I want a stored procedure that filters products by subcategory and price. It would look something like this:

```
CREATE PROCEDURE spProductsByCost
@SubCategoryID int, @Cost money
AS
SELECT ProductID, Name, ProductNumber, StandardCost
FROM Production.Product
WHERE ProductSubcategoryID = @SubCategoryID
AND StandardCost > @Cost
```

Using SQL, the multiple parameters can be passed in a comma-delimited list in the order they were declared:

```
EXECUTE spProductsByCost 1, $1000.00
```

Or the parameters can be passed explicitly by value. If the parameters are supplied by value, it doesn't matter in what order they are supplied:

```
EXECUTE spProductsByCost @Cost = $1000.00, @SubCategoryID = 1
```

If a programmer is using a common data access API, such as ADO or ADO.NET, separate parameter objects are often used to encapsulate these values and execute the procedure in the most efficient manner.

Although views and stored procedures do provide some overlap in functionality, they each have a unique purpose. The view used in the previous example can be used in a variety of settings where it may not be feasible to use a stored procedure. However, if I need to filter records using parameterized values, a stored procedure will allow this but a view will not. So, if the programmer building the product browse screen needs an unfiltered result set and the report designer needs a filtered list of products based on a subcategory parameter, do I create a view or a stored procedure? That's easy, both. Use views as the foundation upon which to build stored procedures. Using the previous example, I select from the view rather than the table:

```
ALTER PROCEDURE spProductCosts
@SubcategoryID int
As
SELECT ProductID, Name, ProductNumber, StandardCost
FROM vProductCosts
WHERE ProductSubcategoryID = @SubcategoryID
```

The benefit may not be so obvious in this simple, one-table example. However, if a procedure were based on the nine-table vEmployeeContactDetail view, the procedure call might benefit from optimizations in the view design and the lower maintenance cost of storing this complex statement in only one object.

Returning Values

The parameter examples shown thus far demonstrate how to use parameters for passing values into a stored procedure. One method to return a value from a procedure is to return a single-column, single-row result set. Although there is probably nothing grossly wrong with this technique, it's not the most effective way to handle simple values. A result set is wrapped in a cursor, which defines the rows and columns, and may be prepared to deal with record navigation and locking. This kind of overkill reminds me of a digital camera memory card I recently ordered from a discount electronics supplier. A few days later, a relatively large box arrived and at first appeared to be filled with nothing more than foam packing peanuts. I had to look carefully to find the postage-size memory card inside.

In addition to passing values into a procedure, parameters can also be used to return values for output. Stored procedure parameters with an `OUTPUT` direction modifier are set to store both input and output values by default. Additionally, the procedure itself is equipped to return a single integer value without needing to define a specific parameter. The return value is also called the return code and defaults to the integer value of 0. Some programming APIs, such as ADO and ADO.NET, actually create

a special output parameter object to handle this return value. Suppose I want to know how many product records there are for a specified subcategory. I'll pass the SubCategoryID using an input parameter and return the record count using an output parameter:

```
CREATE PROCEDURE spProductCountBySubCategory
  @SubCategoryID int,
  @ProdCount int OUTPUT
AS
  SELECT @ProdCount = COUNT(*)
  FROM Production.Product
  WHERE ProductSubcategoryID = @SubCategoryID
```

To test a stored procedure with output parameters in Management Studio, it is necessary to explicitly use these parameters by name. Treat them as if they were variables, but you don't need to declare them. When executing a stored procedure using SQL, the behavior of output parameters can be a bit puzzling because they also have to be passed in. In this example, using the same stored procedure, a variable is used to capture the output parameter value. The curious thing about this syntax is that the assignment seems backward. Remember that the `OUTPUT` modifier affects the direction of the value assignment — in this case, from right to left. The results are shown in Figure 12-15.

```
DECLARE @Out int
EXECUTE spProductCountBySubCategory
@SubCategoryID = 2,
@ProdCount = @Out OUTPUT
SELECT @Out AS ProductCountBySubCategory
```

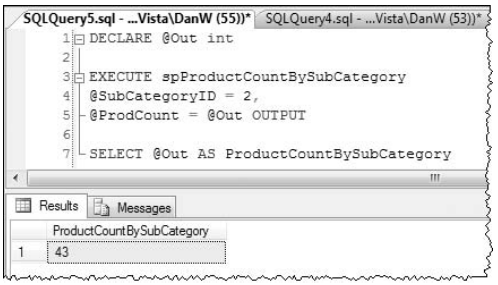

Figure 12-15

It is critical that the `OUTPUT` modifier also be added to the output parameter when it is passed into the stored procedure. If you don't, the stored procedure will still execute, but it will not return any data, as shown in Figure 12-16.

```
DECLARE @Out int
EXECUTE spProductCountBySubCategory
@SubCategoryID = 2,
@ProdCount = @Out --Missing the OUTPUT directional modifier
SELECT @Out AS ProductCountBySubCategory
```

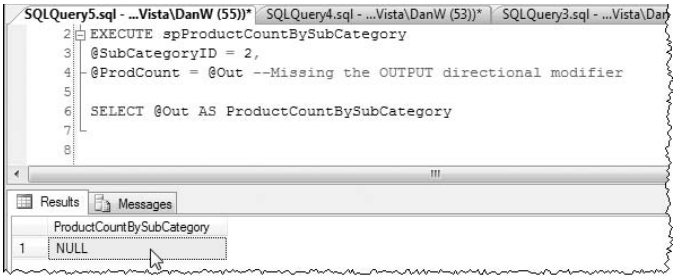

Figure 12-16

There is no practical limit to the number of values that may be returned from a stored procedure. The stated limit is 2,100, including input and output parameters.

If you need to return only one value from the procedure, you can do so without the use of an output parameter. You can use the return code of the procedure as long as the value being returned is an integer. Here is the same stored procedure showing this technique:

```
ALTER PROCEDURE spProductCountBySubCategory
  @SubCategoryID int
AS
  DECLARE @Out int
  SELECT @Out = Count(*)
  FROM Production.Product
  WHERE ProductSubcategoryID = @SubCategoryID
RETURN @Out
```

The `RETURN` statement does two things: it modifies the return value for the procedure from the default value, 0, and it terminates execution so that any statements following this line do not execute. This is significant in cases where there may be conditional branching logic. Typically, the capture of the return value will be done with a programming API. However, you can also execute the return value using Management Studio by setting a variable to the return value of the stored procedure as shown in the following example:

```
DECLARE @Return_Value AS int
EXECUTE @Return_Value =
          spProductCountBySubCategory
                   @SubCategoryID = 2
SELECT @Return_Value
```

Record Maintenance

Using stored procedures to manage the insert, update, and delete operations for each major database entity can drastically reduce the cost of data maintenance tasks down the road. Any program code written to perform record operations should do so using stored procedures and not ad-hoc SQL expressions. As a rule of thumb, when I design a business application, every table that will have records managed through the application interface gets a corresponding stored procedure to perform each of these operations. These procedures are by far the most straightforward in terms of syntax patterns. Although simple, writing this script can be cumbersome due to the level of detail necessary to deal with

all of the columns. Fortunately, Management Studio includes scripting tools that will generate the bulk of the script for you. Beyond creating the fundamental `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements, you need to define and place parameters into your script.

Insert Procedure

The basic pattern for creating a stored procedure to insert records is to define parameters for all non-default columns. In the case of the Product table, the ProductID primary key column will automatically be incremented because it's defined as an identity column. The rowguid and ModifiedDate columns have default values assigned in the table definition, so they will be set if values are not specified. The MakeFlag and FinishedGoodsFlag columns also have default values assigned in the table definition. It may be appropriate to set these values differently for some records. For this reason, these parameters are set to the same default values in the procedure. Several columns are nullable and the corresponding parameters are set to a default value of null. If a parameter with a default assignment isn't provided when the procedure is executed, the default value is used, which is how you create a procedure with optional parameters. Otherwise, all parameters without default values must be supplied:

```
CREATE PROCEDURE spProduct_Insert
       @Name                    nvarchar(50)
     , @ProductNumber           nvarchar(25)
     , @MakeFlag                bit            = 1
     , @FinishedGoodsFlag       bit            = 1
     , @Color                   nvarchar(15)   = Null
     , @SafetyStockLevel        smallint
     , @ReorderPoint            smallint
     , @StandardCost            money
     , @ListPrice               money
     , @Size                    nvarchar(5)    = Null
     , @SizeUnitMeasureCode     nchar(3)       = Null
     , @WeightUnitMeasureCode   nchar(3)       = Null
     , @Weight                  decimal        = Null
     , @DaysToManufacture       int
     , @ProductLine             nchar(2)       = Null
     , @Class                   nchar(2)       = Null
     , @Style                   nchar(2)       = Null
     , @ProductSubcategoryID    smallint       = Null
     , @ProductModelID          int            = Null
     , @SellStartDate           datetime
     , @SellEndDate             datetime       = Null
     , @DiscontinuedDate        datetime       = Null
AS
INSERT INTO Production.Product
  (    Name
     , ProductNumber
     , MakeFlag
     , FinishedGoodsFlag
     , Color
     , SafetyStockLevel
     , ReorderPoint
     , StandardCost
     , ListPrice
     , Size
```

(continued)

(continued)

```
      , SizeUnitMeasureCode
      , WeightUnitMeasureCode
      , Weight
      , DaysToManufacture
      , ProductLine
      , Class
      , Style
      , ProductSubcategoryID
      , ProductModelID
      , SellStartDate
      , SellEndDate
      , DiscontinuedDate  )
VALUES
      ( @Name
      , @ProductNumber
      , @MakeFlag
      , @FinishedGoodsFlag
      , @Color
      , @SafetyStockLevel
      , @ReorderPoint
      , @StandardCost
      , @ListPrice
      , @Size
      , @SizeUnitMeasureCode
      , @WeightUnitMeasureCode
      , @Weight
      , @DaysToManufacture
      , @ProductLine
      , @Class
      , @Style
      , @ProductSubcategoryID
      , @ProductModelID
      , @SellStartDate
      , @SellEndDate
      , @DiscontinuedDate)
```

It's a lot of script but it's not complicated. Executing this procedure in SQL is quite easy. This can be done in comma-delimited fashion or by using explicit parameter names. Because the majority of the fields and corresponding parameters are optional, they can be omitted. Only the required parameters need to be passed; the optional parameters are simply ignored:

```
EXECUTE spProduct_Insert
    @Name               = 'Widget'
  , @ProductNumber      = '987654321'
  , @SafetyStockLevel   = 10
  , @ReorderPoint       = 15
  , @StandardCost       = 23.50
  , @ListPrice          = 49.95
  , @DaysToManufacture  = 30
  , @SellStartDate      = '10/1/04'
```

The procedure can also be executed with parameter values passed in a comma-delimited list. Although the script isn't nearly as easy to read, it is less verbose. Even though this may save you some typing, it often becomes an exercise in counting commas and rechecking the table's field list in the Object Browser until the script runs without error.

```
EXECUTE spProduct_Insert 'Widget', '987654321', 1, 1, Null, 10, 15, 23.50,
49.95, Null, Null, Null, Null, 30, Null, Null, Null, Null, Null, '10/1/04'
```

When using this technique, parameter values must be passed in the order they are declared. Values must be provided for every parameter up to the point of the last required value. After that, the remaining parameters in the list can be ignored.

A useful variation of this procedure may be to return the newly generated primary key value. The last identity value generated in a session is held by the global variable, `@@Identity`. To add this feature, simply add this line to the end of the procedure. This would cause the Insert procedure to return the ProductID value for the inserted record.

```
RETURN @@Identity
```

Of course, if you have already created this procedure, change the `CREATE` keyword to `ALTER`, make changes to the script, and then re-execute it.

Update Procedure

The Update procedure is similar. Usually when I create these data maintenance stored procedures, I write the script for the Insert procedure and then make the modifications necessary to transform the same script into an Update procedure. As you can see, it's very similar:

```
CREATE PROCEDURE spProduct_Update
       @ProductID               int
     , @Name                    nvarchar(50)
     , @ProductNumber           nvarchar(25)
     , @MakeFlag                bit
     , @FinishedGoodsFlag       bit
     , @Color                   nvarchar(15)
     , @SafetyStockLevel        smallint
     , @ReorderPoint            smallint
     , @StandardCost            money
     , @ListPrice               money
     , @Size                    nvarchar(5)
     , @SizeUnitMeasureCode     nchar(3)
     , @WeightUnitMeasureCode   nchar(3)
     , @Weight                  decimal
     , @DaysToManufacture       int
     , @ProductLine             nchar(2)
     , @Class                   nchar(2)
     , @Style                   nchar(2)
     , @ProductSubcategoryID    smallint
     , @ProductModelID          int
     , @SellStartDate           datetime
     , @SellEndDate             datetime
     , @DiscontinuedDate        datetime
```

(continued)

(continued)

```
AS
UPDATE Product
SET    Name                 = @Name
     , ProductNumber        = @ProductNumber
     , MakeFlag             = @MakeFlag
     , FinishedGoodsFlag    = @FinishedGoodsFlag
     , Color                = @Color
     , SafetyStockLevel     = @SafetyStockLevel
     , ReorderPoint         = @ReorderPoint
     , StandardCost         = @StandardCost
     , ListPrice            = @ListPrice
     , Size                 = @Size
     , SizeUnitMeasureCode  = @SizeUnitMeasureCode
     , WeightUnitMeasureCode = @WeightUnitMeasureCode
     , Weight               = @Weight
     , DaysToManufacture    = @DaysToManufacture
     , ProductLine          = @ProductLine
     , Class                = @Class
     , Style                = @Style
     , ProductSubcategoryID = @ProductSubcategoryID
     , ProductModelID       = @ProductModelID
     , SellStartDate        = @SellStartDate
     , SellEndDate          = @SellEndDate
     , DiscontinuedDate     = @DiscontinuedDate
WHERE ProductID = @ProductID
```

The parameter list is the same as the insert procedure with the addition of the primary key, in this case, the ProductID column. However, the defaults have been removed. Defaults are dangerous to use with update procedures because if no value is provided, the procedure will overwrite what is in the table with the default. In our case, most of the defaults are NULL. This would cause any value in the database to be removed and replaced by a NULL if no value were provided.

Delete Procedure

In its basic form, the Delete procedure is very simple. The only necessary parameter for our example is the ProductID column value:

```
CREATE PROCEDURE spProduct_Delete
     @ProductID    int
AS
     DELETE FROM Production.Product
     WHERE ProductID = @ProductID
```

In reality, what needs to be provided is whatever fields uniquely identify the row. This may be a single row, as in the case of the Product table, or a combination of columns in more complex tables.

Remember that deleting rows is not always as easy as it may seem. If the table is referenced with a foreign key constraint, all child records would have to be deleted prior to the deletion of the parent record. For example, the following script will not work:

```
DELETE FROM Production.ProductCategory
WHERE ProductCategoryID = 1
```