Stored Procedures

Understanding batches is a crucial prerequisite to understanding stored procedures, but you shouldn't confuse the two. Batches involve communication between a client program and SQL Server, and stored procedures are objects that exist only on the server. A stored procedure always executes within context of a single batch, but a single batch can contain calls to multiple stored procedures as well as other commands that are not part of stored procedures.

We've actually seen stored procedures in use quite a few times already. For example, we've looked at the output to the system stored procedures *sp_help*, *sp_helpconstraint* and *sp_helpindex*. The stored procedures that you can create are not too different from the system-supplied procedures. After I tell you a bit more about stored procedures, you'll be able to see exactly what the similarities and differences are between the procedures you write and the ones Microsoft supplies with SQL Server.

To create a stored procedure, you take a batch and wrap it inside a CREATE PROCEDURE statement. The procedure's definition can consist of any Transact-SQL commands, with nothing special except for declaring which parameters are passed to the procedure.

To demonstrate how easy it is to create a stored procedure, I've written a simple procedure called *get_author* that takes one parameter, the author ID, and returns the names of any authors that have IDs equal to whatever character string is passed to the procedure:

```
CREATE PROC get_author @au_id varchar(11)
AS
SELECT au_lname, au_fname
FROM authors
WHERE au_id=@au_id
```

This procedure can be subsequently executed with syntax like the following:

```
EXEC get_author '172-32-1176'
```

or

```
EXEC get_author @au_id='172-32-1176'
```

As you can see, you can pass the parameters anonymously by including specific values for them. If the procedure is expecting more than one parameter and you want to pass them anonymously, you must specify their values in the order that the parameters were listed in the CREATE PROCEDURE statement. Alternatively, you can explicitly name the parameters so that the order in which they're passed isn't important.

If executing the procedure is the first statement in the batch, using the keyword EXEC is optional. However, it's probably best to use EXEC (or EXECUTE) so that you won't wind up wondering why your procedure wouldn't execute (only to realize later that it's no longer the first statement of the batch).

In practice, you'll probably want a procedure like the one above to be a bit more sophisticated. Perhaps you'll want to search for author names that begin with a partial ID that you pass. If you choose not to pass anything, the procedure should show all authors rather than return an error message stating that the parameter is missing.

Or maybe you'd like to return some value as a variable (distinct from the result set returned and from the return code) that you can use to check for successful execution. You can do this by passing an output parameter, which has a pass-by-reference capability. Passing an output parameter to a stored procedure is similar to passing a pointer when you call a function in C. Rather than passing a value, you pass the address of a storage area in which the procedure will cache a value. That value is subsequently available to the SQL batch after the stored procedure has executed.

For example, you might want an output parameter to tell you the number of rows returned by a SELECT statement. While the @@ROWCOUNT system function provides this information, it's available for only the last statement executed. If the stored procedure has executed many statements, you must put this value away for safekeeping. An output parameter provides an easy way to do this.

Here's a simple procedure that selects all the rows from the *authors* table and all the rows from the *titles* table. It also sets an output parameter for each table based on @@rowcount. The calling batch retrieves the @@ROWCOUNT values through the variables passed as the output parameters.

```
CREATE PROC count_tables @authorcount int OUTPUT,
@titlecount int OUTPUT
AS
SELECT * FROM authors
SET @authorcount=@@ROWCOUNT
SELECT * FROM titles
SET @titlecount=@@ROWCOUNT
RETURN(0)
```

The procedure would then be executed like this:

```
DECLARE @a_count int, @t_count int
EXEC count_tables @a_count OUTPUT, @t_count OUTPUT
```

TIP

Parameters passed to a routine are treated much like local variables. Variables declared in a stored procedure are always local, so we could have used the same names for both the variables in the procedure and those passed by the batch as output parameters. In fact, this is probably the most common way to invoke them. However, the variables passed into a routine don't need to have the same name as the associated parameters. Even with the same name, they are in fact different variables because their scoping is different.

This procedure returns all the rows from both tables. In addition, the variables *@a_count* and *@t_count* retain the row counts from the *authors* and *titles* tables, respectively. After the EXEC statement inside the batch, we can look at the values in the two variables passed to procedure to see what values they ended up with:

```
SELECT authorcount=@a_count, titlecount=@t_count
```

Here's the output for this SELECT:

```
authorcount    titlecount
-----------    ----------
23             18
```

When you create a stored procedure, you can reference a table, a view, or another stored procedure that doesn't currently exist. In the latter case, you'll get a warning message informing you that a referenced object doesn't exist. As long as the object exists at the time the procedure is executed, all will be fine.

Nested Stored Procedures

Stored procedures can be nested and can call other procedures. A procedure invoked from another procedure can also invoke yet another procedure. In such a transaction, the top-level procedure has a nesting level of 1. The first subordinate procedure has a nesting level of 2. If that subordinate procedure subsequently invokes another stored procedure, the nesting level is 3, and so on, to a limit of 32 nesting levels. If the 32-level limit is reached, a fatal error occurs, the batch is aborted, and any open transaction is rolled back.

The nesting-level limit prevents stack overflows that can result from procedures recursively calling themselves infinitely. The limit allows a procedure to recursively call itself only 31 subsequent times (for a total of 32 procedure calls). To determine how deeply a procedure is nested at runtime, you can select the value of the system function @@NESTLEVEL.

Unlike with nesting levels, SQL Server has no practical limit on the number of stored procedures that can be invoked from a given stored procedure. For example, a main stored procedure can invoke hundreds of subordinate stored procedures. If the subordinate procedures don't invoke other subordinate procedures, the nesting level never reaches a depth greater than 2.

An error in a nested (subordinate) stored procedure isn't necessarily fatal to the calling stored procedure. When you invoke a stored procedure from another stored procedure, it's smart to use a RETURN statement and check the return value in the calling procedure. In this way, you can work conditionally with error situations (as shown in the upcoming factorial example).

Recursion in Stored Procedures

Stored procedures can perform nested calls to themselves, also known as *recursion.* Recursion is a technique by which the solution to a problem can be expressed by applying the solution to subsets of the problem. Programming instructors usually demonstrate recursion by having students write a factorial program using recursion to display a table of factorial values for *0!* through *10!*. Recall that a factorial of a positive integer *n*, written as *n!*, is the product of all integers from 1 through *n*. For example:

```
8!    = 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1 = 40320
```

(Zero is a special case—*0!* is defined as equal to 1.)

We can write a stored procedure that computes factorials, and we can do the recursive programming assignment in Transact-SQL:

```
-- Use Transact-SQL to recursively calculate factorial
-- of numbers between 0 and 12.
-- Parameters greater than 12 are disallowed because the
-- result overflows the bounds of an int.

CREATE PROC factorial @param1 int
AS
DECLARE @one_less int, @answer int
IF (@param1 < 0 OR @param1 > 12)
    BEGIN
        -- Illegal parameter value. Must be between 0 and 12.
        RETURN -1
    END

IF (@param1=0 or @param1=1)
    SELECT @answer=1
ELSE
    BEGIN
        SET @one_less=@param1 - 1
        EXEC @answer=factorial @one_less -- Recursively call itself
        IF (@answer= -1)
            BEGIN
                RETURN -1
            END

        SET @answer=@answer * @param1
        IF (@@ERROR <> 0)
            RETURN -1
    END

RETURN(@answer)
```

Note that when the procedure is initially created, a warning message like the one shown here will indicate that the procedure references a procedure that doesn't currently exist (which is itself in this case):

```
Cannot add rows to sysdepends for the current stored procedure
because it depends on the missing object 'factorial'. The stored
procedure will still be created.
```

Once the procedure exists, we can use it to display a standard factorial:

```
DECLARE @answer int, @param int
SET @param=0
WHILE (@param <= 12) BEGIN
  EXEC @answer = factorial @param
    IF (@answer= -1) BEGIN
        RAISERROR('Error executing factorial procedure.', 16, -1)
        RETURN
    END
PRINT CONVERT(varchar, @param) + '! = ' + CONVERT(varchar(50), @answer)
SET @param=@param + 1
END
```

Here's the output table:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
```

We stopped at *12!* in the *factorial* procedure because *13!* is 6,227,020,800, which exceeds the range of a 32-bit (4-byte) integer. Even if we changed the parameter *@answer* to be type *decimal*, we would still be limited to *12!* because the value included in a RETURN statement must be type *int*.

Here's another version of the procedure that uses an output parameter for the answer and introduces a new variable to hold the returned status of the procedure. You can use a *decimal* datatype with a scale of 0 as an alternative to *int* for integer operations that require values larger than the 4-byte *int* can handle.

```
CREATE PROC factorial @param1 decimal(38,0), @answer decimal(38,0) output
AS
DECLARE @one_less decimal(38,0), @status int

IF (@param1 < 0 OR @param1 > 32)
    BEGIN
        -- Illegal parameter value. Must be between 0 and 32.
        RETURN -1
    END

IF (@param1=0 or @param1=1)
    SET @answer=1
ELSE
    BEGIN
        SET @one_less=@param1 - 1
        EXEC @status=factorial @one_less, @answer output
        -- Recursively call itself
        IF (@status= -1)
            BEGIN
                RETURN -1
            END
        SET @answer=@answer * @param1

        IF (@@ERROR <> 0)
            RETURN -1
    END

RETURN  0
```

To call this procedure, we can use the following code:

```
    DECLARE @answer decimal(38,0), @param int
    SET @param=0
    WHILE (@param <= 32) BEGIN
        EXEC  factorial  @param, @answer output
        IF (@answer= -1)
            BEGIN
                RAISERROR('Error executing factorial procedure.', 16, -1)
                RETURN
            END
        PRINT CONVERT(varchar, @param) + '! = ' + CONVERT(varchar(50), @answer)
        SET @param=@param + 1
    END
```

Even though this procedure has removed the range limit on the result by using a *decimal* datatype to hold the answer, we can only go to *32!* because we'd reach the maximum nesting depth of 32, which includes recursive calls. If we took out the condition *OR @param1 > 32*, this would allow an initial parameter > *32*, which would result in a nesting level that exceeded the maximum. We would then get this error:

```
    Server: Msg 217, Level 16, State 1, Procedure factorial, Line 17
    Maximum stored procedure, function, trigger, or view-nesting level
    exceeded (limit 32).
```

In C, you need to be sure that you don't overflow your stack when you use recursion. Using Transact-SQL shields you from that concern, but it does so by steadfastly refusing to nest calls more than 32 levels deep. You can also watch @@NESTLEVEL and take appropriate action before reaching the hard limit. As is often the case with a recursion problem, you can perform an iterative solution without the restriction of nesting or worries about the stack.

Here's an iterative approach. To illustrate that there's no restriction of 32 levels because it's simple iteration, we'll go one step further, to *33!*. We'll stop at *33!* because *34!* would overflow the precision of a *numeric(38,0)* variable. So even though the iterative approach removes the nesting level limit, we can only go one step further because of the range limitations of SQL Server's exact numeric datatypes:

```
    -- Alternative iterative solution does not have the restriction
    -- of 32 nesting levels
    CREATE PROC factorial2 @param1 int, @answer NUMERIC(38,0) OUTPUT
    AS
    DECLARE @counter int
    IF (@param1 < 0 OR @param1 > 33)
        BEGIN
            RAISERROR ('Illegal Parameter Value. Must be between 0 and 33',
                16, -1)
            RETURN -1
        END

    SET @counter=1 SET @answer=1

    WHILE (@counter < @param1 AND @param1 <> 0 )
        BEGIN
            SET @answer=@answer * (@counter + 1)
            SET @counter=@counter + 1
        END

    RETURN
    GO

    DECLARE @answer numeric(38, 0), @param int
    SET @param=0
    WHILE (@param <= 32)
        BEGIN
            EXEC factorial2 @param, @answer OUTPUT
            PRINT CONVERT(varchar(50), @param) + '! = '
                + CONVERT(varchar(50), @answer)
            SET @param=@param + 1
        END
```

And here's the output :

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
 13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 51090942171709440000
22! = 1124000727777607680000
23! = 25852016738884976640000
24! = 620448401733239439360000
25! = 15511210043330985984000000
26! = 403291461126605635584000000
27! = 10888869450418352160768000000
28! = 304888344611713860501504000000
29! = 8841761993739701954543616000000
30! = 265252859812191058636308480000000
31! = 8222838654177922817725562880000000
32! = 263130836933693530167218012160000000
33! = 8683317618811886495518194401280000000
```

Because this factorial procedure is really producing a single valued result, we could have created it as a user-defined function. I'll show you how to do that a little later in this chapter.

Stored Procedure Parameters

Stored procedures take parameters, and you can give parameters default values. If you don't supply a default value when you create the procedure, an actual parameter will be required when the procedure is executed. If you don't pass a required parameter, you'll get an error like the following, which occurs when I try to call the factorial procedure without a parameter:

```
Server: Msg 201, Level 16, State 3, Procedure factorial, Line 0
Procedure 'factorial' expects parameter '@param1', which was not supplied.
```

You can pass values by explicitly naming the parameters or by furnishing all the parameter values anonymously but in correct positional order. You can also use the keyword DEFAULT as a placeholder when you pass parameters. You can also pass NULL as a parameter (or define it as the default). Here's a simple example:

```
CREATE PROCEDURE pass_params
@param0 int=NULL,    -- Defaults to NULL
@param1 int=1,       -- Defaults to 1
@param2 int=2        -- Defaults to 2
AS
SELECT @param0, @param1, @param2
GO

EXEC pass_params           -- PASS NOTHING - ALL Defaults
(null)    1    2

EXEC pass_params 0, 10, 20    -- PASS ALL, IN ORDER
0    10    20

EXEC pass_params @param2=200, @param1=NULL
-- Explicitly identify last two params (out of order)
(null)    (null)    200

EXEC pass_params 0, DEFAULT, 20
-- Let param1 default. Others by position.
0    1    20
```

Note that if you pass parameters by value, you can't leave any gaps. That is, you can't pass only the first and third, using syntax like this:

```
EXEC pass_params 0,, 20
-- You'll get a syntax error here
```

Also, be aware of the difference between leaving a parameter value unspecified, using the keyword DEFAULT, and passing in an explicit NULL. In the first two cases, the procedure must have a default defined for the parameter in order to avoid an error. If you pass an explicit NULL, you will not get an error, but if there is a default value for the parameter, it will not be used. NULL will be the "value" used by the procedure.

Wildcards in Parameters

If you have a parameter that is of type *character*, you might want to pass an incomplete string as a parameter and use it for pattern matching. Consider this procedure that accepts an input parameter of type *varchar* and returns the title of any book from the *titles* table in the *pubs* database for which the *title_id* matches the input string:

```
CREATE PROC gettitles
@tid varchar(6) = '%'
-- the parameter default means that if no parameter
--   is specified, the procedure will return all the
--   titles
AS

SELECT title
```

```
FROM titles
WHERE title_id LIKE @tid

IF @@rowcount = 0
    PRINT 'There are no titles matching your input'

RETURN
```

We can call this procedure in a number of different ways. You should be aware that in many cases, SQL Server is very smart about datatype conversion, and the following is an acceptable statement, even without quotes around the character parameter:

```
EXEC gettitles bu1032
```

The parser breaks the command into individual units of meaning, called *tokens*, by using spaces and other nonalphanumeric characters as separators. Since the parameter here is completely alphanumeric, the parser recognizes it as a single token and can then assume that it is a parameter of the procedure. The procedure expects a parameter of type *varchar*, so it just interprets the value it receives as a *varchar*.

If your parameter includes any nonalphanumeric characters, such as dashes or spaces, the parser will not be able to deal with it appropriately and you'll need to help it by enclosing such parameters in quotes or square brackets. So note what happens if we call the *gettitles* procedure with a *%* in the parameter:

```
EXEC gettitles  bu%

RESULT:
Server: Msg 170, Level 15, State 1, Line 1
Line 1: Incorrect syntax near '%'.
```

This error is not generated by the stored procedure but by the parser before the procedure is ever called. The parser is unable to determine that the *bu%* needs to be treated as a single token. If there are quotes around the parameter, the parser is satisfied and the procedure can be executed. Because the query inside the procedure uses LIKE instead of =, the pattern matching can be done and we get back all the titles for which the *title_id* starts with *bu*:

```
EXEC gettitles  'bu%'

RESULTS:
title
---------------------------------------------------------
The Busy Executive's Database Guide
Cooking with Computers: Surreptitious Balance Sheets
You Can Combat Computer Stress!
Straight Talk About Computers

(4 row(s) affected)
```

If you want your procedure to handle wildcards in parameters, you need to make sure not only that the search condition uses LIKE, but also that the type of parameter is *varchar* instead of *char*. You might not think that this would be necessary here because all of the *title_id* values are exactly six characters long. But suppose that in this *gettitles* example we had declared *@tid* to be *char(6).* This would mean that the procedure would need a parameter of exactly six characters. If we had passed in the three characters *bu%*, SQL Server would have had to pad the parameter with spaces (*'bu% '*) to get the full six characters. Then, when the comparison was made in the WHERE clause of the procedure's SELECT statement, SQL Server would look for *title_id* values that start with *bu*, are followed by anything (because of the %), and end with three spaces! Because none of the *title_id* values in the *titles* table end with three spaces, no rows would be returned.

The default of *%* in the procedure means that if no parameter is specified, the procedure will compare the *title_id* with *%* and return all the rows in the table. This is a useful technique to make the procedure more versatile. Many of our system procedures work this way; for example, if you execute *sp_helpdb* with a parameter, that parameter is considered the name of a database and you'll get back details about that database. With no parameter, *sp_helpdb* will return information about *all* your databases.

However, this technique works only for character string parameters because that is the only datatype that allows wildcards. What if I wanted to write a procedure to find all books with a particular value for *price*? If no price is specified, I want the procedure to return the names of all the books.

One solution is to define a parameter with a default of NULL. I could write the procedure as two completely separate SELECT statements, one for when the parameter has a non-null value and one without a WHERE clause for when the parameter is NULL. However, there is another way to do this in a single SELECT statement:

```
CREATE PROC gettitle_by_price
@cost money = NULL
AS
SELECT price, title
FROM titles
WHERE price = ISNULL(@cost, price)
RETURN
```

This procedure assigns a default value of NULL to the parameter, which prevents an error from being generated if no parameter is actually passed in. However, only a single SELECT statement is needed. The ISNULL function will end up comparing a price to itself if the parameter is NULL, and all values for price will be equal to themselves.

There is one little gotcha here, which is that the result from executing this procedure with no parameter is not exactly the same as executing a SELECT without a WHERE clause. A SELECT with no WHERE clause will return ALL the rows. The SELECT with the ISNULL in the WHERE clause will return all rows in which the value of *price* is equal to the parameter or equal to the same value for *price*. But what if *price* has no value? In the *titles* table, two rows have a nonvalue, in other words NULL, for the *price*, and will not be returned from the above procedure. In this situation, even setting ANSI_NULLS to OFF will not help. We'll try to find the rows WHERE price = price, and because NULL is never considered equal to NULL, the rows with NULL for price will not be returned. One alternative would be to write the WHERE clause like this:

```
WHERE ISNULL(price, 0) = ISNULL(@cost, ISNULL(price, 0))
```

If we write the procedure with this condition, the results we will get from executing the procedure with no parameter will be the same as executing a SELECT without a WHERE clause; that is, all the rows will be returned. However, the drawback to this solution is that SQL Server might not be able to optimize this query and find a good execution plan. As we'll see in Chapter 16, if a column is embedded in a function call or other expression, the optimizer is very unlikely to consider using any indexes on that column. That means if our table has an index on the *price* column and we actually do pass in a value for *price*, SQL Server probably won't choose to use that index to find the rows in the table with the requested price.