Stored Procedures

A stored procedure is a database object that contains one or more SQL statements. In the following you will learn how to create and use stored procedures.

Stored procedures can return data in any of the following ways (although returning any data is optional):

- One or more recordsets (returned by having SELECT statements in the procedure)
- Messages, returned by using the PRINT command, or RAISERRROR command for errors.
- A return integer status code that can be used to indicate the resulting state of the code (for instance, whether successful or not)
- By modifying values of OUTPUT arguments

Stored procedures are often used to create "interfaces" to the database for program developers.

The first time a procedure is executed, each SQL statement it contains is compiled and executed to create an execution plan. Then the procedure is stored in compiled from within the database. For each subsequent execution, the SQL statements are executed without compilation, because they're precompiled. This makes the execution of a stored procedure faster than the execution of an equivalent SQL script.

To execute, or call a stored procedure, you use the EXEC statement. If the EXEC statement is the first line in a batch, you can omit the EXEC keyword and just code the procedure name (since this might become hard to read and understand it is recommended to have the EXEC keyword).

The Syntax to CREAT PROC Statement

```
CREATE {PROC | PROCEDURE} procedure_name
[parameter declaration]
AS
      SQL_Statement

RETURN
```

The following is a simple stored procedure with a single SELECT statement.

```
CREATE PROC spInvoiceReport
AS
      SELECT VendorName, InvoiceNumber,InvoiceDate, InvoiceTotal
      FROM Invoices JOIN Vendors ON Invoices.VendorID=Vendors.VendorID
      WHERE InvoiceTotal-CreditTotal-PaymentTotal>0
      ORDER BY VendorName;
RETURN
```

Once the code is run successfully the following statement is needed to get the result.

```
EXEC spInvoiceReport
```

- You can call a stored procedure from within another stored procedure. You can even call a stored procedure from within itself. This technique is called recursive call or recursion, and is seldom used in SQL programming.
- One of the advantages of using procedures is that application programmers and end users don't need to know the structure of the database or how to code SQL.
- Another advantage of using procedures is that they can restrict and control access to a database. If you use procedures in this way, you can prevent both accidental errors and malicious damage.
- When naming stored procedures it is recommended to start the name with sp to help distinguish it from other database objects.
- CREAT PROC must be the first and only statement in the batch. If you have other statements before CREAT PROC insert a GO at the end of those statements.

The following examples creates a procedure to copy a table. The IF statements are added to avoid getting errors if the proc or the table already exists.

```
IF OBJECT_ID ('spCopyInvoice') IS NOT NULL
      DROP PROC spCopyInvoice
GO

CREATE PROC spCopyInvoices
AS
      IF OBJECT_ID('InvoiceCopy') IS NOT NULL
            DROP TABLE InvoiceCopy
      SELECT *
      INTO InvoiceCopy
      FROM Invoices;
```

How to Declare and Work with Parameters

@parameter identifies a name and datatype for an argument that will be passed to the procedure. A default value can also be provided for the parameter in case the calling client does not provide one.

Stored procedures provide for two different types of parameters: input parameters and output parameters. An input parameter is passed to the stored procedure from the calling program. An output parameter is returned to the calling program from the stored procedure. You identify an output parameter with the OUTPUT keyword. If this keyword is omitted, the parameter is assumed to be an input parameter.

You can declare an input parameter so it requires a value or so its value is optional. The value of a required parameter must be passed to the stored procedure from the calling program or an error occurs. The value of an optional parameter doesn't need to be passed from the calling program. You identify an optional parameter by assigning a default value to it. Then, if a value isn't passed from the calling program, the default value is used.

```
@DateVar smalldatetime                  --Input parameter
```

```
@VendorVar varchar(40) = NULL          --Optional Input parameter
@InvTotal money OUTPUT                 --Output parameter
```

The following example creates a procedure that uses an input and an output parameter.

```
CREATE PROC spInvTotal
     @DateVar smalldatetime,
     @InvTotal money OUTPUT
AS
SELECT @InvTotal = SUM(InvoiceTotal)
FROM Invoices
WHERE InvoiceDate >= @DateVar;
```

Here is another example with an optional parameter:

```
CREATE PROC spInvTotal2
     @DateVar smalldatetime = NULL
AS
IF @DateVar IS NULL
     SELECT @DateVar = MIN (InvoiceDate) FROM Invoices
SELECT SUM(InvoiceTotal)
FROM Invoices
WHERE InvoiceDate >= @DateVar;
```

How to Call Procedures with Parameters

To pass parameter values to a stored procedure, you code the values in the EXEC statement after the procedure name. You can pass parameters to a stored procedure either by position or by name. The first EXEC below, passes parameters by position. When you use this technique, you don't include the names of the parameters. Instead, the parameters are listed in the same order as they appear in the CREATE PROC statement (this is the most common way).

```
DECLARE @MyInvTotal money
EXEC spInvTotal '2016-02-10', @MyInvTotal OUTPUT
PRINT @MyInvTotal
```

You can pass parameters by name, as the following example shows.

```
DECLARE @MyInvTotal money
EXEC spInvTotal @DateVar = '2016-02-10', @InvTotal = @MyInvTotal
OUTPUT
PRINT @MyInvTotal
```

How to Work with Return Values

In addition to passing output parameters back to the calling program, stored procedures also pass back a return value. By default, this value is zero. However you can use a RETURN statement to return another number.
In the following example spInvCount returns a count of the number of invoices that meet the conditions specified by the input parameters. Since the procedure uses a RETURN statement to return an integer value, there's no need to use an output parameter.

```
CREATE PROC spInvCount
      @DateVar smalldatetime = NULL,
      @VendorVar varchar(40) = '%'
AS
IF @DateVar IS NULL
      SELECT @DateVar = MIN (InvoiceDate) FROM Invoices

DECLARE @InvCount int;

SELECT @InvCount = COUNT(InvoiceID)
FROM Invoices JOIN Vendors ON Invoices.VendorID = Vendors.VendorID
WHERE (InvoiceDate >= @DateVar ) AND (VendorName LIKE @VendorVar);

RETURN @InvCount
```

To EXEC the stored procedure

```
DECLARE @InvCount int;
EXEC @InvCount = spInvCount '2016-02-01', 'P%';
PRINT 'Invoice count: ' + CONVERT (varchar,@InvCount);
```

- If a stored procedure needs to return a single integer value, many programmers prefer using a RETURN statement. However, if a stored procedure needs to return other types of data, or if it needs to return multiple values, then a RETURN statement won't work.
- The RETURN statement immediately exits the procedure and returns an optional integer value to the calling program.
- To use the return value in the calling program, you must declare a variable to store its value. Then, you code that variable name followed by an equal sign and the name of the procedure in the EXEC statement.

To Alter an Existing Stored Procedure

Substitute the word ALTER for CREATE in the syntax. This basically just overwrites the existing stored procedure with the new definition.

Deleting a Stored Procedure

To drop a stored procedure:

```
DROP PROCEDURE spInvCount
```

Example

Here is an example stored procedure that can be used to update employee salaries in a given employee classification by a given percentage:

```
CREATE PROCEDURE IncreaseSalary
      @WhatPercent float = 0.0,
```

```
      @WhichJobClass int = 1
AS

/* View matching employees before update */
SELECT *
FROM Employees
WHERE JobClass = @WhichJobClass

/* Do update */
UPDATE Employees
SET Salary=Salary * (1.0 + @WhatPercent )
WHERE JobClass = @WhichJobClass

/* View matching employees after update */
SELECT *
FROM Employees
WHERE JobClass = @WhichJobClass

RETURN
```