# Creating, compiling, and using stored procedures

Referenced from SQL Magazine (http://www.sqlmag.com)

A stored procedure is compiled code that you can call from within T-SQL statements or from client applications. SQL Server runs the code in the procedure and then returns the results to the calling application. Using stored procedures is efficient for several reasons. First, SQL Server has already parsed, optimized, and compiled stored procedures, so they run quickly without needing to repeat these steps each time. Also, stored procedures run on the SQL Server, using the power of the server and reducing the load on the client (which might be a much less powerful computer). Finally, using stored procedures reduces network traffic. Instead of the SQL Server sending all the data to the client and having the client run a query, the client sends a request to the server to run the procedure. The server returns only the result data set, which is usually a lot smaller than the full data set. Once a stored procedure has run, it remains in memory, so the next user can run it without incurring the overhead of loading it into memory. SQL Server 6.5 and earlier releases require multiple copies of the procedure in memory if more than one user will be running the procedure at the same time. SQL Server 7.0 improved on this situation by letting multiple users, each with a different execution context (including parameters and variables), share one copy of the procedure in memory. You can use stored procedures to enforce a level of consistency in your client applications. If all the client applications use the same stored procedures to update the database, the code base is smaller and easier to maintain, and you run less risk of deadlocks because everyone is updating tables in the same order.

Stored procedures enhance security, too, because you can give the users only EXECUTE permission on the stored procedures, while restricting access to the tables and not allowing the users any direct update privileges. When other users run a stored procedure, they run it as if they had the permissions of the user who created the query. You can do almost anything in a stored procedure, including CREATE DEFAULT, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, and CREATE VIEW. You can create tables—both permanent and temporary—but any temporary objects you create within a stored procedure vanish when the stored procedure is complete. And you can nest stored procedures up to 32 levels deep, so that one procedure can call another, which calls a third, and so on.

## *Definitions*

In SQL Server Books Online (BOL) and elsewhere, you might see references to several types of stored procedures, but there are only two primary types. A stored procedure is T-SQL code that SQL Server has parsed, optimized, and compiled. An extended stored procedure is a DLL (typically written in C or C++) that leverages the integration of SQL Server and Windows 2000/NT to make OS-level calls and perform functions that are beyond T-SQL's scope. Both types can accept input parameters and return output values, error codes, and status messages. Within the two primary types of stored procedure, you'll find several others.

Local stored procedure: a standard stored procedure, executed on the local computer. You define stored procedures within a database and they become local objects within that database. You can call local stored procedures from another database by giving the full name, including the database name, owner name, and procedure name. You can also call them from other applications, including common client applications such as Microsoft Access and Visual Basic (VB) applications. Typically, you use the Query Analyzer to create local stored procedures. (You could use the Enterprise Manager, but it's not very helpful—you still have to know how to write the code, and it's easier to test from the Query Analyzer.) Temporary stored procedure: just like a local stored procedure, except that the name starts with a # symbol for a private temporary stored procedure and a ## for a global temporary stored procedure. A private temporary procedure is good only within the session it was created in; a global procedure can be seen and run from other sessions. Note the difference in terminology between temporary tables and temporary procedures. Temporary tables can be local or global, but a "local" stored procedure is any in the current database, so we use the word private to denote stored procedures that are limited to the current session.

Remote stored procedures: a standard stored procedure, run on a remote computer. As Einstein might have pointed out, to an observer on the remote computer, this is actually a local stored procedure that has been

activated from another computer. The stored procedure exists on the remote computer; you just send across the network the set of instructions to run it. In SQL Server 7.0, distributed queries officially replaced remote stored procedures, but the ideas are the same—one server asks another to run a stored procedure and return the results to the calling server.

System stored procedure: a stored procedure that you create in the Master database and that is available to any other database. System stored procedure names start with sp_; the term "system stored procedures" is usually understood to mean the stored procedures Microsoft supplies with SQL Server. You can create your own system stored procedure simply by creating a stored procedure in Master and prefixing it with sp_. However, Microsoft recommends that you don't use sp_ when you name a stored procedure. If you do, regardless of where you created it, SQL Server looks first in Master, then in the database you specified in your calling string, and finally (if you didn't specify a database) in the local database with the Database Owner (DBO) as the owner. Yes, SQL Server checks Master first, even if you supply the database name. Not only is this inefficient if the procedure is local, but a procedure with the same name might exist in Master, leading to confusion.

An extended stored procedure is a DLL that is coded in a programming language other than T-SQL. An extended stored procedure's name begins with xp_. Microsoft supplies with SQL Server some extended stored procedures, such as xp_readmail, which handles reading email messages on behalf of the sp_processmail stored procedure. One especially versatile extended stored procedure is xp_cmdshell, which executes a command string as an OS command shell. (Essentially, you can run from xp_cmdshell any command that you can run from the Win2K/NT command line.)

## *Creating a Stored Procedure*

To create a stored procedure, you must be the DBO or systems administrator (sa), or be a member of the db_ddladmin role. Users with the permission to grant permissions can grant other users permission to create procedures using T-SQL commands. You can't grant this permission to specific users from the Enterprise Manager interface; you need to place them in the db_ddladmin role instead. When you create the procedure, you must have permission to perform all the steps listed in the query. These steps might include accessing other tables or performing inserts, updates, and deletes.

When you create a procedure, you can create it only in the current database. So you need to specify only the procedure name, not the database name or even the owner name. The exception to this rule is temporary stored procedures, which are created in tempdb. Procedure names must be unique within a database, but you can use the same procedure name in different databases. If you want a procedure to be available in every database but don't want to make it a system stored procedure in Master, add it to the Model database. It will appear in every new database you create thereafter.

You can't combine the CREATE PROCEDURE statement with any other statement in the same batch. That restriction sounds minor—you might think that you can just put in a GO statement and move on to the next part of the script. However, you can't put a GO statement in the middle of a procedure: As soon as it reaches a GO, the parser treats that as the end of the procedure and compiles everything up to that point.

The syntax for creating a stored procedure is simple:

```
CREATE PROC procedure_name
[@parameter datatype] [= default] [OUPUT],
[@parameter datatype] [= default] [OUPUT],
….

WITH {RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION}
AS
T-SQL statement…….
```

You specify input parameters as `@parameter_name`, and you must define a data type for each parameter. Optionally, you can specify a default value for any parameters; the default must be a constant or NULL.

Procedures can contain up to 1024 parameters; these are local to the procedure so you can use the same parameter names for other procedures without risk of interference. If you want a parameter to return a value, you must specify the parameter as an OUTPUT parameter when you create the procedure. You don t necessarily always have to ask for the parameter as a returned parameter, but unless you have defined it as a potential output parameter, you can t ask for a parameter to return a value when you execute the procedure. Parameters can act as both input and output parameters, so a simple procedure could have only one input parameter, with the value being modified and passed back as an output parameter. The WITH ENCRYPTION option prevents others from reverse engineering your procedures by looking at the code in the syscomments table. It also prevents you from looking at the code because there is no way to ask for the code to be unencrypted. SQL Server can unencrypt it, because it can recompile the code when necessary, including when you upgrade to a new release of SQL Server. But you can t supply a key or password to unencrypt the code, so keep a copy of your source code somewhere secure. If you need to change the definition of a stored procedure, you can do so by rerunning the CREATE PROCEDURE statement, changing CREATE PROCEDURE to ALTER PROCEDURE. SQL Server 7.0 introduced the ability to ALTER database objects, so you can change objects without having to drop and recreate them as you had to do in previous releases. Dropping the object also removes all permissions on that object, so when you recreate a dropped object, you also have to rebuild the permissions. (Scripting the permissions on an object before you drop it is always a good idea.)

## *The Creation Process*

When you create a stored procedure, SQL Server places an entry in the sysobjects table for the database, listing the new object. SQL Server parses the T-SQL code and checks it for syntax errors, then stores the procedure code in the syscomments table (in its encrypted form, if you chose that option). A process called Delayed Name Resolution lets you create stored procedures and refer to objects, such as tables, that don t yet exist. The parser doesn t give you an error because it assumes that the object referenced will exist by the time you execute the query.

## *Running a Procedure for the First Time*

When you execute stored procedure for the first time, the query optimizer builds an execution plan for the procedure, then compiles the plan and uses it to run the procedure. SQL Server 2000 and 7.0 don t store this execution plan permanently. The plan remains in memory unless your recompile options specify otherwise. The procedure cache is an area of memory where SQL Server keeps stored procedures and cached queries. SQL Server 6.5 and earlier releases required you to configure how much of the available memory to allocate to data cache and how much to procedure cache. SQL Server 2000 and 7.0 allocate the memory dynamically to each cache as SQL Server requires.

## *Running the Procedure Again*

Once the procedure is in memory, other client applications can use it without any action on their part—they simply run the procedure, and if it is found in memory, they use it. If the procedure isn t in memory, SQL Server must reoptimize and compile it. Most SQL Servers with adequate memory keep frequently run procedures in the cache. But if memory use becomes a concern, SQL Server can drop some procedures from memory. It uses a sophisticated algorithm to decide which to drop and which to keep, giving preference to the most frequently used procedures, but taking into account the effort necessary to recompile a procedure if it were flushed from the cache. In other words, the algorithm calculates whether to drop one large procedure that hasn t been used for a long time, or several small procedures that are used occasionally but not frequently. Users can share a copy of the procedure in memory as long as their environment—the server, database, and connection settings—is identical. Different connection settings require different copies of the procedure in memory, so try to standardize connection settings (SET options such as ANSI PADDING, for example). And here s a heads-up for database designers:

Users can t share a copy of a procedure if a referenced object requires name resolution. Such a situation can occur when two objects have the same name but different owners: for example, the sales and engineering departments might each have added a table called "budget," with different owners. Or a programmer might have made a copy of the employees table, and own the copy but not the original.

Having rules in place about object naming and ownership—ideally all objects have unique names and are owned by the DBO—will prevent this problem.

## Recompile Options

One benefit of stored procedures is that they remain in memory and can be reused until the server is restarted, so they could be in memory for months. In some situations, you'll want to recompile a procedure. For example, you might have a procedure that produces radically differing results every time you run it, such as a procedure to return a list of customer names and addresses for a range of ZIP codes. If you run this procedure on a few ZIP codes in New York, you might get back several thousand names. But if you run it on a larger range of ZIP codes in Wyoming, you might get only two or three names. For such a stored procedure, consider putting the keywords `WITH RECOMPILE` in the `CREATE PROC` code. SQL Server will then discard the compiled execution plan every time the procedure is run, and recompile it again for the next user. You might usually run a stored procedure with certain parameters—for example, if most of your business is in New York, you expect a large number of names per ZIP code. When you need to send a mailing to Wyoming, you can run the query again, using the `EXECUTE procname WITH RECOMPILE` option. SQL Server discards the plan in the procedure cache and compiles a new one with the atypical parameters, so your procedure runs with the query optimized for this mailing list. As a courtesy to the next user, you should run the query again, using a typical data set and the `WITH RECOMPILE` option. If you don't, your atypical plan will sit in memory, and the next time someone runs the procedure, SQL Server will use it. You might also consider recompiling your procedure if you add an index to the table. Recompiling gives the query optimizer a chance to calculate whether the new index might be useful. The command to use here is `sp_recompile`. You can specify a procedure name for the recompile, or you can supply a table name or view name. If you specify a table or view, SQL Server will recompile all procedures that reference that table or view the next time they run.

## Automatic Recompile

BOL says that "SQL Server automatically recompiles stored procedures and triggers when it is advantageous to do so." Actually, all SQL Server does is kick the procedure out of the cache, so it has to recompile the procedure the next time it runs. SQL Server recomputes the statistics on an index when there have been a significant number of changes to the data. With new statistics, recompiling (and therefore reoptimizing) the query makes sense. If you drop an index that a procedure used, the procedure will need recompiling. Any time you rebuild indexes to reclaim space in the database, SQL Server recompiles the procedure with the new index statistics. So it's unlikely that any procedure will remain in memory indefinitely. In fact, by not storing optimized query plans, SQL Server 2000 and 7.0 look for opportunities to recompile with the latest statistics.

## Temporary Stored Procedures vs. Cached Queries

As I mentioned, you can create a temporary stored procedure by starting the procedure name with # or ##, and these procedures are built in tempdb. A private procedure can be used only in the session where it was created; a global procedure can be used by anyone with the right permissions, but it vanishes when the session it was created in is disconnected.

Fortunately, SQL Server lets anyone running the procedure at the time complete the procedure. Also, if you create stored procedures in tempdb directly, you don't need to start the name with # or ##. You just run the following command from your database

```
EXECUTE tempdb.dbo.procname
```

and the stored procedure remains in tempdb until the server restarts, even if you log off. If you don't want to create temporary stored procedures in tempdb, you can create the procedure in your database, use it, then drop it.

SQL Server 2000 and 7.0 can cache queries, so SQL Server can spot a query being repeated. It can also detect a query being run over and over with just one different parameter—say someone in sales is calling up customer data and the only difference in each iteration is the customerID. SQL Server can take this

query, treat customerID as a parameter, and use the same execution plan with different customerID values. That way, SQL Server avoids having to parse, optimize, and compile every time. (This process is sometimes called autoparameterization.) You also have the option of using the sp_executesql system stored procedure to tell SQL Server that you're going to keep sending it the same query with different values. This option is great if you have a loop in your code and use it to step through a list of updates to a table one at a time.

So if SQL Server can cache queries and discard them when it is done with them, what's the difference between a temporary stored procedure and a cached query? Not a lot, really. Both are parsed, optimized, compiled, used, and dropped. The biggest difference is that the cached query uses space in your database, and the temporary procedure is built in tempdb. That distinction might be not be important unless you're working on a server with many other users who are competing for space in tempdb. Microsoft recommends the sp_executesql approach, and I agree because of the reduced possibility of contention in tempdb.

## *Automatic Stored Procedure Execution*

SQL Server has a little-known feature called automatic stored procedure execution, which lets you specify stored procedures to run when the server starts—it's like an autoexec.bat file for SQL Server. To make a procedure run at startup, run the `sp_procoption` procedure with the syntax `sp_procoption procname startup true`. You can use true or on to make the procedure run at startup and false or off to remove it from the list of procedures to run at startup. (In SQL Server 6.5, you run sp_makestartup and sp_unmakestartup to add procedures to or remove them from the autostart list. These procedures don't exist in later releases.) Be careful about adding multiple procedures because all the procedures will start at once. If you need the procedures to run in a sequence, build one procedure that calls all the others in the right order, then make that one an autostart procedure.

## *Stored Procedure Guidelines*

You might prefer to use a prefix of usp_ for user stored procedures, but there's no accepted industry-wide naming standard. Microsoft recommends limiting each stored procedure to executing one task. If you need to perform multiple tasks, you can create one procedure that calls all the others in the correct sequence. If you've used the tasks in the sequence elsewhere, modularizing the code makes sense. But there's no reason not to have a sequence of tasks in one procedure. You might encounter problems, though, if you build branching logic into the procedure or if you have a procedure that performs one of several different tasks depending on the input parameters. If you run the procedure the first time with a set of parameters that cause it to activate branch B, SQL Server can't optimize the code for branches A and C. And if you use branching logic, you load into memory code that you might not use very often. So for a straight-through sequence that SQL Server processes from start to finish, you might be better off writing it with more steps in fewer procedures. For branching logic, having each branch as a separate procedure and calling them as needed might make more sense.

Stored procedures are a powerful part of SQL Server. They offer benefits in both security and performance. Any client application you use will almost certainly run better if you add some stored procedures running on the server.