PART I

The Basics

CHAPTER 1

Stored Procedure Primer

Today, average software development practices are becalmed in a windless sea of code-and-fix programming—a kind of flat-earth approach to software development that was proven ineffective 20 years ago.

—Steve McConnell[1]

Working from the assumption that the human brain learns by associating new data with what it already knows, we'll spend this chapter building a base framework onto which we can assemble the knowledge conveyed by the remainder of the book. We'll touch on the topics covered in the book's other chapters, but we'll save the details for the chapters themselves. I'm assuming that you know some basic Transact-SQL with which we can associate these high-level concepts. We'll spend the remainder of the book filling in the details and expanding on what we cover here.

This chapter serves to prime the discussion on SQL Server stored procedure programming. It will tell you what a stored procedure is, how stored procedures are often used, and why and how you should use them. It will also jumpstart the treatment of Transact-SQL as a full-fledged programming language. If I could have you take one thing away from reading this book, it would be that Transact-SQL programming is very much like any other type of programming: It requires the same attention to detail, the same craftsmanship, and the same software engineering skill to do well.

What Is a Stored Procedure?

A Transact-SQL stored procedure is a set of T-SQL code that is stored in a SQL Server database and compiled when used. You create this set of code

1. McConnell, Steve. *After the Gold Rush*. Redmond, WA: Microsoft Press, 1998. Page 91.

using the CREATE PROCEDURE command. You can use most Transact-SQL commands in a stored procedure; however, some commands (such as CREATE PROCEDURE, CREATE VIEW, SET SHOWPLAN_TEXT, SET SHOWPLAN_ALL, and so forth) must be the first (or only) statement in a command batch, and therefore aren't allowed in stored procedures. Most Transact-SQL commands behave the same in a stored procedure as they do in a command batch, but some have special capabilities or exhibit different behavior when executed within the context of a stored procedure. Listing 1–1 shows a simple stored procedure (only the code from the CREATE PROCEDURE line down to the ensuing GO actually constitutes the stored procedure):

Listing 1–1 A simple stored procedure.

```
Use Northwind
GO
IF OBJECT_ID('dbo.ListCustomersByCity') IS NOT NULL
  DROP PROC dbo.ListCustomersByCity
GO
CREATE PROCEDURE dbo.ListCustomersByCity @Country nvarchar(30)='%'
AS
SELECT City, COUNT(*) AS NumberOfCustomers
FROM Customers
WHERE Country LIKE @Country
GROUP BY City
GO
EXEC dbo.ListCustomersByCity
```

Stored Procedure Advantages

Although you can do most of the things a stored procedure can do with simple ad hoc Transact-SQL code, stored procedures have a number of advantages over ad hoc queries, including

- Execution plan retention and reuse
- Query autoparameterization
- Encapsulation of business rules and policies
- Application modularization
- Sharing of application logic between applications
- Access to database objects that is both secure and uniform
- Consistent, safe data modification

- Network bandwidth conservation
- Support for automatic execution at system start-up

I'll touch on each of these as we go along.

Creating a Stored Procedure

As I've said, you use the Transact-SQL CREATE PROCEDURE command to create stored procedures. All that really happens when you create a procedure is that its syntax is checked and its source code is inserted into the syscomments system table. Generally, object names referenced by a procedure aren't resolved until it's executed. In SQL Server parlance, this is known as *deferred name resolution*.

"Syscomments" is a misnomer. The table doesn't store comments per se; it stores source code. The name is a vestige from the pre-7.0 days and was so named because it stored the optional source code to stored procedures (and other objects), whereas sysprocedures stored a pseudo-compiled version (a kind of normalized query tree) of the procedures themselves. This is no longer true, and the sysprocedures table no longer exists. Syscomments is now the sole repository for stored procedures, views, triggers, user-defined functions (UDFs), rules, and defaults. If you remove an object's source code from syscomments, you can no longer use that object.

Deferred Name Resolution and an Interesting Exception

Before we go further, it's worth mentioning that there's an interesting exception to SQL Server's default deferred name resolution behavior. Run the code in Listing 1–2 in Query Analyzer:

Listing 1–2 SQL Server doesn't allow you to include more than one CREATE TABLE statement for a given temporary table in the same stored procedure.

```
CREATE PROC testp @var int
AS
IF @var=1
  CREATE TABLE #temp (k1 int identity, c1 int)
ELSE
  CREATE TABLE #temp (k1 int identity, c1 varchar(2))
INSERT #temp DEFAULT VALUES
SELECT c1 FROM #temp
GO
```

The syntax contained in the stored procedure is seemingly valid, yet we get this message when we run it:

```
Server: Msg 2714, Level 16, State 1, Procedure testp, Line 6
There is already an object named '#temp' in the database.
```

Why? Obviously @var can't be both equal to one and not equal to one, right? To get a little closer to the answer, change the temporary table reference to a permanent table, like the one shown in Listing 1–3.

Listing 1–3 Changing the table to a permanent table works around the temporary table limitation.

```
CREATE PROC testp @var int
AS
IF @var=1
  CREATE TABLE tempdb..temp (k1 int identity, c1 int)
ELSE
  CREATE TABLE tempdb..temp (k1 int identity, c1 varchar(2))
INSERT #temp DEFAULT VALUES
SELECT c1 FROM #temp
GO
```

This procedure is created without error. What's happening here? Why does SQL Server care whether the table created is a temporary or a permanent table? And why does it matter *now*—before the procedure is even executed and before the value of @var can be known?

What appears to be happening is that SQL Server resolves CREATE TABLE references to temporary tables *before* inserting the procedure into syscomments—an apparent vestige from the pre-7.0 days when object references were resolved when a procedure was first created. The same limitation applies to variable declarations and, therefore, to the **table** data type. You can't declare a variable more than once in a single stored procedure, even if the declarations reside in mutually exclusive units of code. This differs from how permanent tables are handled, and is the reason the code in Listing 1–3 runs without error. It appears that, beginning with SQL Server 7.0, deferred name resolution was enabled for permanent tables, but not for temporary ones. Whatever the case, you can't execute code like that shown in Listing 1–2, so here's a workaround (Listing 1–4):

Listing 1–4 Including one CREATE TABLE statement, but two sets of ALTER TABLE statements, works around the problem.

```
CREATE PROC testp @var int
AS
CREATE TABLE #temp (k1 int identity)
IF @var=1
  ALTER TABLE #temp ADD c1 int
ELSE
  ALTER TABLE #temp ADD c1 varchar(2)
INSERT #temp DEFAULT VALUES
EXEC('SELECT c1 FROM #temp')
GO
```

This technique creates the table just once, then alters it to match the Data Definition Language (DDL) specification (spec) we want to end up with. Note the use of EXEC() to select the column we added with ALTER TABLE. The use of dynamic SQL is necessary because the newly added column isn't immediately visible to the procedure that added it. We're forced to create and execute an ad hoc query string to access it by name. (Note that you *can* reference the column indirectly—for example, through SELECT * or via an ordinal value in an ORDER BY clause, just not by name).

Another disadvantage of this approach is that it mixes DDL (the CREATE and ALTER statements) and Data Modification Language (DML; the INSERT and SELECT). Because of this, the procedure's execution plan must be recompiled when the INSERT is encountered (the temporary table's schema information (info) has changed since the original execution plan was formulated). Any stored procedure that creates a temporary table, then processes it further, will cause a plan recompile because the table's schema info did not exist when the execution plan was first created; however, the procedure in Listing 1–4 causes an *additional* recompile to occur because it alters this schema, then processes the table further. Particularly with large procedures in high-throughput environments, this can cause performance problems as well as blocking and concurrency issues because a compile lock is taken out on the stored procedure while the execution plan is being recompiled. Listing 1–5 presents a workaround that doesn't require the use of dynamic T-SQL:

Listing 1–5 A workaround for the temporary table creation problem.

```
CREATE PROCEDURE testp4
AS
```

```
INSERT #temp DEFAULT VALUES
SELECT c1 FROM #temp
GO

CREATE PROC testp3
AS
CREATE TABLE #temp (k1 int identity, c1 varchar(2))
EXEC dbo.testp4
GO

CREATE PROC testp2
AS
CREATE TABLE #temp (k1 int identity, c1 int)
EXEC dbo.testp4
GO

CREATE PROC testp @var int
AS
IF @var=1
EXEC dbo.testp2
ELSE
EXEC dbo.testp3
GO
```

Although this technique alleviates the need for EXEC(), it also forces us to completely reorganize the stored procedure. In fact, we're forced to break the original procedure into four separate routines and call the fourth one redundantly from the second and third routines. Why? First, instead of having two CREATE TABLE statements for the same temporary table in one procedure—which, as we've discovered, isn't supported—we moved each CREATE TABLE to its own procedure. Second, because a temporary table is automatically dropped as soon as it goes out of scope, we can't simply create it, then return to the top-level routine and add rows to it or SELECT from it. We have to do that either in one of the procedures that created it or in a common routine that they call. We chose the latter, so procedures two and three call a fourth routine that takes care of inserting the row into the temporary table and selecting the **c1** column from it. (Because objects created in a procedure are visible to the procedures it calls, the fourth routine can "see" the table created by its caller.) This approach works, but is far from optimal. Think about how complex this would get for a really large procedure. Breaking it into multiple, distinct pieces may not be practical. Still, it avoids the necessity of having to create and execute an ad hoc T-SQL string and should generally perform better than that approach.

Listing a Stored Procedure

Assuming the object is not encrypted, you can list the source code to a procedure, view, trigger, UDF, rule, or default object using the sp_helptext system procedure. An example is included in Listing 1–6:

Listing 1–6 sp_helptext lists the source for a stored procedure.

```
EXEC dbo.sp_helptext 'ListCustomersByCity'
Text
------------------------------------------------------------------------
CREATE PROCEDURE dbo.ListCustomersByCity @Country nvarchar(30)='%'
AS
SELECT City, COUNT(*) AS NumberOfCustomers
FROM Customers
WHERE Country LIKE @Country
GROUP BY City
```

Permissions and Limitations

Only members of the sysadmin, db_owner, or db_ddladmin role (or those explicitly granted CREATE PROC permission by a member of the appropriate role) can execute CREATE PROCEDURE.

The maximum stored procedure size is 128MB. The maximum number of parameters a procedure may receive is 1,024.

Creation Tips

Include a comment header with each procedure that identifies its author, purpose, creation date and revision history, the parameters it receives, and so forth. A common technique is to place this comment block either immediately before or just after the CREATE PROC statement itself (but before the rest of the procedure) to ensure that it's stored in syscomments and can be viewed from tools like Enterprise Manager and Query Analyzer's Object Browser. The system stored procedure that follows, sp_object_script_comments, generates comment headers for stored procedures, views, and similar objects (Listing 1–7):

Listing 1–7 You can use sp_object_script_comments to generate stored procedure comment headers.

```
USE master
GO
IF OBJECT_ID('dbo.sp_object_script_comments') IS NOT NULL
```

```
  DROP PROC dbo.sp_object_script_comments
GO
CREATE PROCEDURE dbo.sp_object_script_comments
                    -- Required parameters
                    @objectname sysname=NULL,
                    @desc sysname=NULL,

                    -- Optional parameters
                    @parameters varchar(8000)=NULL,
                    @example varchar(8000)=NULL,
                    @author sysname=NULL,
                    @workfile sysname='', -- Force workfile to be generated
                    @email sysname='(none)',
                    @version sysname=NULL,
                    @revision sysname='0',
                    @datecreated smalldatetime=NULL,
                    @datelastchanged smalldatetime=NULL
/*

Object: sp_object_script_comments
Description: Generates comment headers for SQL scripts

Usage: sp_object_script_comments @objectname='ObjectName',
@desc='Description of object",@parameters='param1[,param2...]'

Returns: (None)

$Workfile: sp_object_script_comments.sql $

$Author: Khen $. Email: khen@khen.com

$Revision: 1 $

Example: sp_object_script_comments @objectname='sp_who', @desc='Returns a
list of currently running jobs', @parameters=[@loginname]

Created: 1992-04-03. $Modtime: 1/4/01 8:35p $.

*/
AS

IF (@objectname+@desc) IS NULL GOTO Help

PRINT '/*'
```

```
PRINT CHAR(13)
EXEC sp_usage @objectname=@objectname,
              @desc=@desc,
              @parameters=@parameters,
              @example=@example,
              @author=@author,
              @workfile=@workfile,
              @email=@email,
              @version=@version, @revision=@revision,
              @datecreated=@datecreated, @datelastchanged=@datelastchanged
PRINT CHAR(13)+'*/'

RETURN 0

Help:
EXEC dbo.sp_usage @objectname='sp_object_script_comments',
       @desc='Generates comment headers for SQL scripts',
       @parameters='@objectname=''ObjectName'',
       @desc=''Description of object",@parameters=''param1[,param2...]''',
       @example='sp_object_script_comments @objectname=''sp_who'',
       @desc=''Returns a list of currently running jobs'',
       @parameters=[@loginname]',
       @author='Ken Henderson',
       @workfile='sp_object_script_comments.sql',
       @email='khen@khen.com',
       @version='3', @revision='1',
       @datecreated='19920403', @datelastchanged='19990701'
RETURN -1
GO
EXEC dbo.sp_object_script_comments
```

This procedure generates stored procedure comment headers by calling the sp_usage procedure included later in the chapter. It can be executed from any database by any procedure. To use sp_object_script_comments, simply pass it the required parameters, and it will create a fully usable comment block that identifies a procedure or other type of object and spells out its usage and key background info. You can copy this block of text and paste it into the header of the routine itself and—*voila!*—you've got a nicely formatted, informative comment block for your code.

In shops with lots of stored procedure code, it's common to locate each stored procedure in its own script and to store each script in a version control or source code management system. Many of these systems support special tags (these are known as *keywords* in Visual SourceSafe [VSS], the source code

management system that I use) that you can embed in T-SQL comments. Through these tags, you allow the source code management system to automatically insert revision information, the name of the person who last changed the file, the date and time of the last change, and so on. Because the tags are embedded in comments, there's no danger that these changes will break your code. Basically, you're just allowing the system to take care of some of the housekeeping normally associated with managing source code. Many of the stored procedures listed in this book include tags recognized by VSS in their headers (these tags begin and end with $). See Chapter 4 for more information.

Allow the passing of a single help parameter such as '/?'—or no parameters—to return an informational message telling the caller how to use the procedure. Place the section that generates this usage information at the end of the procedure to keep it out of the way and to locate it consistently from procedure to procedure. An ideal way to do this is to set up and call a separate procedure that accepts parameters indicating usage information and returns it in a uniform format. Here's a stored procedure that does just that (Listing 1–8):

Listing 1–8 You can use sp_usage to generate stored procedure usage info.

```
USE master
GO
IF OBJECT_ID('dbo.sp_usage') IS NOT NULL
  DROP PROC dbo.sp_usage
GO
CREATE PROCEDURE dbo.sp_usage
                      -- Required parameters
                      @objectname sysname=NULL,
                      @desc sysname=NULL,
                      -- Optional parameters
                      @parameters varchar(8000)=NULL,
                      @returns varchar(8000)='(None)',
                      @example varchar(8000)=NULL,
                      @workfile sysname=NULL,
                      @author sysname=NULL,
                      @email sysname='(none)',
                      @version sysname=NULL,
                      @revision sysname='0',
                      @datecreated smalldatetime=NULL,
                      @datelastchanged smalldatetime=NULL
/*

Object: sp_usage
```

```
Description: Provides usage information for stored procedures and
descriptions of other types of objects

Usage: sp_usage @objectname='ObjectName', @desc='Description of object'
              [, @parameters='param1,param2...']
              [, @example='Example of usage']
              [, @workfile='File name of script']
              [, @author='Object author']
              [, @email='Author email']
              [, @version='Version number or info']
              [, @revision='Revision number or info']
              [, @datecreated='Date created']
              [, @datelastchanged='Date last changed']

Returns: (None)

$Workfile: sp_usage.sql $

$Author: Khen $. Email: khen@khen.com

$Revision: 7 $

Example: sp_usage @objectname='sp_who', @desc='Returns a list of currently
running jobs', @parameters=[@loginname]

Created: 1992-04-03. $Modtime: 1/04/01 8:38p $.

*/
AS
SET NOCOUNT ON
IF (@objectname+@desc IS NULL) GOTO Help

PRINT 'Object: '+@objectname
PRINT 'Description: '+@desc

IF (OBJECTPROPERTY(OBJECT_ID(@objectname),'IsProcedure')=1)
OR (OBJECTPROPERTY(OBJECT_ID(@objectname),'IsExtendedProc')=1)
OR (OBJECTPROPERTY(OBJECT_ID(@objectname),'IsReplProc')=1)
OR (LOWER(LEFT(@objectname,3))='sp_') BEGIN -- Special handling for system
procedures
  PRINT CHAR(13)+'Usage: '+@objectname+' '+@parameters
  PRINT CHAR(13)+'Returns: '+@returns
END
-- $NoKeywords: $ -- Prevents the keywords below from being expanded in VSS
```

```
IF (@workfile IS NOT NULL)
  PRINT CHAR(13)+'$Workfile: '+@workfile+' $'
IF (@author IS NOT NULL)
  PRINT CHAR(13)+'$Author: '+@author+' $. Email: '+@email
IF (@version IS NOT NULL)
  PRINT CHAR(13)+'$Revision: '+@version+'.'+@revision+' $'
IF (@example IS NOT NULL)
  PRINT CHAR(13)+'Example: '+@example
IF (@datecreated IS NOT NULL) BEGIN -- Crop time if it's midnight
  DECLARE @datefmt varchar(8000), @dc varchar(30), @lc varchar(30)
  SET @dc=CONVERT(varchar(30), @datecreated, 120)
  SET @lc=CONVERT(varchar(30), @datelastchanged, 120)
  PRINT CHAR(13)+'Created: '+CASE
DATEDIFF(ss,CONVERT(char(8),@datecreated,108),'00:00:00') WHEN 0 THEN
LEFT(@dc,10) ELSE @dc END
+'. $Modtime: '+CASE
DATEDIFF(ss,CONVERT(char(8),@datelastchanged,108),'00:00:00') WHEN 0 THEN
LEFT(@lc,10) ELSE @lc END+' $.'
END

RETURN 0

Help:
EXEC dbo.sp_usage @objectname='sp_usage',            -- Recursive call
          @desc='Provides usage information for stored procedures and
descriptions of other types of objects',
          @parameters='@objectname=''ObjectName'', @desc=''Description of
object''
              [, @parameters=''param1,param2...'']
              [, @example=''Example of usage'']
              [, @workfile=''File name of script'']
              [, @author=''Object author'']
              [, @email=''Author email'']
              [, @version=''Version number or info'']
              [, @revision=''Revision number or info'']
              [, @datecreated=''Date created'']
              [, @datelastchanged=''Date last changed'']',
          @example='sp_usage @objectname=''sp_who'',
          @desc=''Returns a list of currently running jobs'',
          @parameters=[@loginname]',
          @author='Ken Henderson',
          @workfile='sp_usage.sql',
          @email='khen@khen.com',
          @version='3', @revision='1',
```

```
          @datecreated='4/3/92', @datelastchanged='7/1/99'
RETURN -1

GO
EXEC dbo.sp_usage
```

By passing in the appropriate parameters, you can use sp_usage to report usage info for any procedure. Sp_usage even calls itself for that very purpose (that's why we receive the warning message: "Cannot add rows to sysdepends for the current stored procedure because it depends on the missing object 'sp_usage.' The stored procedure will still be created."). Because Transact-SQL doesn't support subroutines, sp_usage uses a GOTO label to place the help message at the end of the procedure. This approach allows code at the start of the procedure to check for invalid parameter values and to jump quickly to the usage routine if necessary.

Set the QUOTED_IDENTIFIER and ANSI_NULLS options before you execute CREATE PROCEDURE (in its own command batch) because they're reset to the values they had at the time the procedure was created when it's executed (their values are stored in the **status** column of the procedure's row in sysobjects). This change lasts only for the duration of the procedure; afterward, they're restored to whatever they were before you executed the procedure. Setting QUOTED_IDENTIFIER or ANSI_NULLS *inside* a stored procedure has no effect on the execution of the stored procedure. To see how this works, run the code in Listing 1–9 in Query Analyzer:

Listing 1–9 SET ANSI_NULLS has no effect inside a stored procedure.

```
USE tempdb
GO
SET ANSI_NULLS ON
GO
CREATE PROC testn
AS
SET ANSI_NULLS OFF
DECLARE @var int
SET @var=NULL
SELECT * FROM Northwind..Customers WHERE @var=NULL
GO
EXEC testn
```

(Results abridged)

```
CustomerID CompanyName                              ContactName
---------- ---------------------------------------- ----------------
(0 row(s) affected)
```

If ANSI_NULLS is actually off at the time of the SELECT, as the SET command inside the procedure specifies, the SELECT should return all the rows in the Northwind Customers table. As you can see, this is not what happens. Now change the SET ANSI_NULLS command that precedes the CREATE PROCEDURE to turn ANSI null handling OFF, and rerun the procedure. You should see all the rows in the Customers table listed.

Set environmental options (e.g., NOCOUNT, LOCK_TIMEOUT, and so on) that materially affect the procedure early on. It's a good practice to set them at the very start of the procedure so that they stand out to other developers.

Avoid broken ownership chains when dealing with stored procedures and the objects they reference. Try to ensure that the owner of a stored procedure and the owner of the objects it references are the same. The best way to do this is by specifying the dbo user as the owner of every object you create. Having multiple objects with the same name but different owners adds an unnecessary layer of indirection to the database that's almost always more trouble than it's worth. While perhaps useful during the development phase of a project, it's definitely something you should avoid on production servers.

When used within a stored procedure, certain commands require the objects they reference to be owner qualified (an object reference is said to be owner qualified when the object name is prefixed with the name of the owner and a period) if the procedure is to be executed by users other than the owner. These commands are

- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- TRUNCATE TABLE
- CREATE INDEX
- DROP INDEX
- UPDATE STATISTICS
- All DBCC commands

Don't use the **sp_** prefix for anything but system procedures. Because of the confusion it can cause, avoid creating procedures in user databases with the **sp_** prefix. Also, don't create nonsystem procedures in the master database. If

a procedure is not a system procedure, it's likely that you don't need to put it in the master database in the first place.

Include USE *dbname* at the top of creation scripts for procedures that must reside in a specific database. This ensures that they end up where they belong and alleviates having to remember to set the current database context before executing the script.

Keep each stored procedure as simple and modular as possible. Ideally, a stored procedure will accomplish a single task or a small group of closely related tasks.

As a rule, SET NOCOUNT ON should be the first statement in every stored procedure you create because it minimizes network traffic between SQL Server and client applications. Setting NOCOUNT on disables DONE_IN_PROC messages—the messages SQL Server normally sends to the client indicating the number of rows affected by a T-SQL statement. Because these messages are very rarely used, eliminating them conserves network bandwidth without really giving up any functionality and can speed up applications considerably. Note that you can disable DONE_IN_PROC messages for the entire server via a trace flag (3640) and for a particular user session via the **sp_configure 'user options'** command. (In rare circumstances, disabling DONE_IN_PROC messages can cause problems with some applications—for example, some older versions of Microsoft Access and certain ill-behaved OLEDB providers).

Create a procedure using the WITH ENCRYPTION option if you want to keep its source code from being viewable by users. Don't delete it from syscomments. Doing so will render the procedure inaccessible and you'll have to drop and recreate it.

Altering Stored Procedures

Just as you create stored procedures using the CREATE PROCEDURE command, you alter them with ALTER PROCEDURE. The advantage of using ALTER PROCEDURE to change a stored procedure is that it preserves access permissions, whereas CREATE PROCEDURE doesn't. A key difference between them is that ALTER PROCEDURE requires the use of the same encryption and recompile options as the original CREATE PROCEDURE statement. If you omit or change them when you execute ALTER PROCEDURE, they'll be omitted or changed permanently in the actual procedure definition.

A procedure can contain any valid Transact-SQL command except these: CREATE DEFAULT, CREATE FUNCTION, CREATE PROC, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER, CREATE VIEW, SET SHOWPLAN_TEXT, and SET SHOWPLAN_ALL. These commands must reside in their own command batches, and, therefore, can't be part of a stored procedure. Procedures *can* create databases, tables, and indexes, but not other procedures, defaults, functions, rules, schemas, triggers, or views.

TIP: You can work around this limitation—the inability to construct most other kinds of objects from within a stored procedure—by constructing a T-SQL string and executing it via sp_executesql or the EXEC() function, as shown in Listing 1–10:

Listing 1–10 You can create procedures, views, UDFs, and other objects from within stored procedures by using sp_executesql and EXEC().

```
CREATE PROC test AS
DECLARE @sql nvarchar(100)
SET @sql=N'create proc dbo.test2 as select ''1'''
EXEC dbo.sp_executesql @sql
EXEC dbo.test2
GO
EXEC dbo.test
```

(Results)

```
Cannot add rows to sysdepends for the current stored procedure because it
depends on the missing object 'dbo.test2'. The stored procedure will still
be created.

----
1
```

The warning message is due to the fact that the test2 procedure doesn't exist when the test procedure is first created. You can safely ignore it.

Executing Stored Procedures

Although executing a stored procedure can be as easy as listing it on a line by itself in a T-SQL command batch, you should make a habit of prefixing all stored procedure calls with the EXEC keyword, like this:

```
EXEC dbo.sp_who
```

Stored procedure calls without EXEC must be the first command in a command batch. Even if this were the case initially, inserting additional lines before the procedure call at some point in the future would break your code.

You should also be sure to owner-qualify procedure calls ("dbo" in the previous example). Omitting the owner from a procedure call causes SQL Server to momentarily place a compile lock on the procedure because it cannot locate it immediately in the procedure cache. This lock is released once the procedure-sans-owner is located in the cache, but can still cause problems in high-throughput environments. Owner-qualifying objects is simply a good habit to get into. It's one of those things you can do to save yourself problems down the road.

INSERT and EXEC

The INSERT command supports calling a stored procedure to supply rows for insertion into a table. Listing 1–11 shows how:

Listing 1–11 You can use INSERT…EXEC to save a stored procedure's output in a table.

```
CREATE TABLE #locks (spid int, dbid int, objid int, objectname sysname
NULL, indid int, type char(4), resource char(15), mode char(10), status
char(6))
INSERT #locks (spid, dbid, objid, indid, type, resource, mode, status)
EXEC dbo.sp_lock
SELECT * FROM #locks
DROP TABLE #locks
```

This is a handy way of trapping the output of a stored procedure in a table so that you can manipulate it or retain it for later use. Prior to the advent of cursor OUTPUT parameters, this was the only way to perform further work on a stored procedure's result set within Transact-SQL.

Note that INSERT…EXEC works with extended procedures that return result sets as well. A simple example is shown in Listing 1–12:

Listing 1–12 INSERT…EXEC works with extended procedures as well.

```
CREATE TABLE #cmd_result (output varchar(8000))
INSERT #cmd_result
EXEC master.dbo.xp_cmdshell 'TYPE C:\BOOT.INI'
SELECT * FROM #cmd_result
DROP TABLE #cmd_result
```

Execution Plan Compilation and Execution

When you execute a stored procedure for the first time, it's compiled into an execution plan. This plan is not compiled into machine code or even byte codes, but is pseudo-compiled in order to speed execution. By "pseudo-compiled" I mean that object references are resolved, join strategies and indexing selections are made, and an efficient plan for executing the work that the procedure is to carry out is rendered by the SQL Server query optimizer. The optimizer compares a number of potential plans for performing the procedure's work and selects the one it thinks will cost the least in terms of total execution time. It bases this decision on a number of factors, including the estimated I/O cost associated with each plan, the CPU cost, the memory requirements, and so on.

Once an execution plan has been created, it's stored in the procedure cache for future execution. This cache grows and contracts as necessary to store execution plans for the stored procedures and ad hoc queries executed by the server. SQL Server balances the need to supply adequate memory to the procedure cache with the server's other memory requirements, such as providing adequate resources for the data cache. Obviously, memory taken up by cached execution plans can't be used to cache data, so the server manages this carefully. Caching execution plans in memory saves the optimizer from having to construct a new plan each time a procedure is executed, and can improve performance dramatically.

Monitoring Execution

You can inspect the manner in which SQL Server compiles, stores, and runs execution plans using SQL Server's Profiler utility. To observe what happens when you create and run a procedure, follow these steps:

1. Start the Query Analyzer utility, connect to your server, and load the stored procedure script from Listing 1–1 (you can find the complete script on the CD accompanying this book).
2. Start the Profiler utility. You should find it in your Microsoft SQL Server **Start|Programs** folder.
3. Click the New Trace button and connect to your server.
4. On the Events page, remove every event class from the list on the right except the SQL:BatchStarting event class in the TSQL group.
5. Add every event class in the Stored Procedures group on the left except the SP:StmtStarting and SP:StmtComplete events. (A trace template file that includes these events, BasicTrace.TDF, is on the CD accompanying this book).

6. Click the Run button at the bottom of the Trace Properties dialog.
7. Return to Query Analyzer and run the script.
8. Return to Profiler and click the Stop Selected Trace button. You should see something like the following in the events window:

(Results abridged)

```
EventClass            TextData
--------------------- ----------------------------------
SQL:BatchStarting     Use Northwind
SQL:BatchStarting     IF OBJECT_ID('dbo.ListCustomersByCi
SQL:BatchStarting     CREATE PROCEDURE dbo.ListCustomersB
SQL:BatchStarting     EXEC dbo.ListCustomersByCity
SP:CacheMiss
SP:CacheMiss
SP:CacheInsert
SP:Starting           EXEC dbo.ListCustomersByCity
SP:Completed          EXEC dbo.ListCustomersByCity
```

The trace output begins with four separate T-SQL command batches. Because the commands are separated by the GO batch terminator, each executes as a separate T-SQL batch. The last batch is the call to the stored procedure via the EXEC command. This call is responsible for the events that follow.

Note the SP:CacheInsert event immediately before the SP:Starting event. In conjunction with the SP:CacheMiss events, this tells us that ListCustomersByCity wasn't in the procedure cache when it was called, so an execution plan was compiled for it and inserted into the cache. The final two events in the trace, the SP:Starting and SP:Completed events, indicate that once the execution plan for the stored procedure was inserted into the cache, it was executed.

To see what happens when a procedure is executed directly from the cache, follow these steps:

1. Click the Start Selected Trace button to restart the trace.
2. Return to Query Analyzer, highlight the EXEC line in the query, and run it by itself.
3. Return to Profiler and stop the trace. You should see something like this:

(Results abridged)

```
EventClass          TextData
------------------- --------------------------
SQL:BatchStarting   EXEC dbo.ListCustomersByCity
SP:ExecContextHit
```

```
SP:Starting          EXEC dbo.ListCustomersByCity
SP:Completed         EXEC dbo.ListCustomersByCity
```

The ExecContextHit event tells us that an executable version of the stored procedure was found in the cache. Note the absence of the SP:CacheMiss and CacheInsert events. This tells us that the execution plan that was created and inserted into the cache when we ran the stored procedure the first time is reused when we run it a second time.

Execution Plans

When SQL Server runs an execution plan, each step of the plan is processed and dispatched to an appropriate internal manager process (e.g., the T-SQL manager, the DDL and DML managers, the transaction manager, the stored procedure manager, the utility manager, the ODSOLE manager, and so on). SQL Server calls these managers repeatedly until it has processed all the steps in the execution plan.

Execution plans are never stored permanently. The only portion of a stored procedure that is stored on disk is its source code (in syscomments). Because they're cached in memory, cycling the server disposes of all current execution plans (as does the DBCC FREEPROCCACHE() command).

SQL Server automatically recreates a stored procedure's execution plan when

- The procedure's execution environment differs significantly from its creation environment (see Environmental Issues discussed later in the chapter for more information)
- The sysobjects **schema_ver** column changes for any of the objects the procedure references. The **schema_ver** and **base_schema_ver** columns are updated any time the schema information for a table changes. This includes column additions and deletions, data type changes, constraint additions and deletions, as well as rule and default bindings.
- The statistics have changed for any of the objects the procedure references. This means that the auto-update statistics and auto-create statistics events can cause stored procedure recompilation.
- An index is dropped that was referenced by the procedure's execution plan
- A copy of the procedure's execution plan is not available in the cache. Execution plans are removed from the cache to make room for new plans using a Least Recently Used (LRU) algorithm.

- Certain other specialized circumstances occur, such as when a temporary table is modified a fixed number of times, when DDL and DML statements are interleaved, and when the sp_configure system procedure is called (sp_configure calls DBCC FREEPROCCACHE)

During the earlier discussion on creating procedures and SQL Server's limitation regarding having multiple CREATE TABLE statements for a temporary table in a single procedure, I mentioned that the ad hoc code approach (Listing 1–4) forces the procedure's execution plan to be recompiled while it's running. To see this for yourself, restart the trace we've been using and rerun the stored procedure from that query. You should see something like the following in Profiler:

```
EventClass           TextData
-------------------- ----------------------------------------
SQL:BatchStarting    exec testp 2
SQL:StmtStarting     exec testp 2
SP:ExecContextHit
SP:Starting          exec testp 2
SQL:StmtStarting     -- testp CREATE TABLE #temp (k1 int identity)
SQL:StmtStarting     -- testp IF @var=1
SQL:StmtStarting     -- testp ALTER TABLE #temp ADD c1 varchar(2)
SP:Recompile
SP:CacheMiss
SP:CacheMiss
SP:CacheInsert
SQL:StmtStarting     -- testp ALTER TABLE #temp ADD c1 varchar(2)
SQL:StmtStarting     -- testp INSERT #temp DEFAULT VALUES
SP:Recompile
SP:CacheMiss
SP:CacheMiss
SP:CacheInsert
SQL:StmtStarting     -- testp INSERT #temp DEFAULT VALUES
SQL:StmtStarting     -- testp EXEC('SELECT c1 FROM #temp')
SQL:StmtStarting     -- Dynamic SQL SELECT c1 FROM #temp
SP:Completed         exec testp 2
```

Notice that not one, but two SP:Recompile events occur during the execution of the procedure: one when the ALTER TABLE is encountered (this statement refers to the temporary table created by the procedure, forcing a recompile) and another when the INSERT is encountered (this statement accesses the newly modified temporary table schema, again forcing a recom-

pile). Assuming you've captured the SQL:StmtStarting or SP:StmtStarting event class in the trace, you'll typically see an SP:Recompile event enveloped in two identical StmtStarting events: The first one indicates that the statement began to be executed, but was put on hold so that the recompile could happen; the second indicates that the statement is actually executing now that the recompile has completed. This starting/stopping activity can have a serious impact on the time it takes the procedure to complete. It's worth pointing out again: Creating a temporary table within a procedure that you then process in other ways will cause the procedure's execution plan to be recompiled (one way to avoid temporary tables is to use local **table** variables instead). Moreover, interleaving DDL and DML within a procedure can also cause the plan to be recompiled. Because it can cause performance and concurrency problems, you want to avoid causing execution plan recompilation when you can.

Another interesting fact that's revealed by the trace is that the execution plan for the dynamic T-SQL string the procedure creates and executes is not cached. Note that there's no CacheMiss, CacheInsert, CacheHit, or ExecContextHit event corresponding to the dynamic SQL query near the end of the trace log. Let's see what happens when we change the EXEC() call to use sp_executesql instead (Listing 1–13):

Listing 1–13 You can use sp_executesql rather than EXEC() to execute dynamic T-SQL.

```
USE tempdb
GO
drop proc testp
GO
CREATE PROC testp @var int
AS
CREATE TABLE #temp (k1 int identity)
IF @var=1
  ALTER TABLE #temp ADD c1 int
ELSE
  ALTER TABLE #temp ADD c1 varchar(2)
INSERT #temp DEFAULT VALUES
EXEC dbo.sp_executesql N'SELECT c1 FROM #temp'
GO
exec testp 2
```

When you execute the procedure, you should see trace output like this:

```
EventClass         TextData
------------------ ------------------------------------------------------
SQL:BatchStarting  exec testp 2

SQL:StmtStarting   exec testp 2

SP:CacheMiss
SP:CacheMiss
SP:CacheInsert
SP:Starting        exec testp 2

SQL:StmtStarting   -- testp CREATE TABLE #temp (k1 int identity)

SQL:StmtStarting   -- testp IF @var=1

SQL:StmtStarting   -- testp ALTER TABLE #temp ADD c1 varchar(2)

SP:Recompile
SP:CacheMiss
SP:CacheMiss
SP:CacheInsert

SQL:StmtStarting   -- testp ALTER TABLE #temp ADD c1 varchar(2)

SQL:StmtStarting   -- testp INSERT #temp DEFAULT VALUES

SP:Recompile
SP:CacheMiss
SP:CacheMiss
SP:CacheInsert
SQL:StmtStarting   -- testp INSERT #temp DEFAULT VALUES
SQL:StmtStarting   -- testp EXEC dbo.sp_executesql N'SELECT c1 FROM #temp'
SP:CacheMiss
SP:CacheMiss
SP:CacheInsert     SELECT c1 FROM #temp
SQL:StmtStarting   SELECT c1 FROM #temp
SP:Completed       exec testp 2
```

Note the SP:CacheInsert event that occurs for the dynamic SELECT statement now that we are calling it via sp_executesql. This indicates that the execution plan for the SELECT statement has been inserted into the cache so that it can be reused later. Whether it actually *will* be reused is another matter, but

at least the possibility exists that it can be. If you run the procedure a second time, you'll see that the call to sp_executesql itself generates an ExecContextHit event rather than the CacheMiss event it causes the first time around. By using sp_executesql, we've been able to use the procedure cache to make the procedure run more efficiently. The moral of the story is this: sp_executesql is generally a more efficient (and therefore faster) method of executing dynamic SQL than EXEC().

Forcing Plan Recompilation

You can also force a procedure's execution plan to be recompiled by

- Creating the procedure using the WITH RECOMPILE option
- Executing the procedure using the WITH RECOMPILE option
- Using the sp_recompile system procedure to "touch" any of the tables the procedure references (sp_recompile merely updates sysobjects' schema_ver column)

Once an execution plan is in the cache, subsequent calls to the procedure can reuse the plan without having to rebuild it. This eliminates the query tree construction and plan creation that normally occur when you execute a stored procedure for the first time, and is the chief performance advantage stored procedures have over ad hoc T-SQL batches.

Automatically Loading Execution Plans

A clever way of loading execution plans into the cache at system start-up is to execute them via an autostart procedure. Autostart procedures must reside in the master database, but they can call procedures that reside in other databases, forcing those procedures' plans into memory as well. If you're going to take this approach, creating a single autostart procedure that calls the procedures you want to load into the cache rather than autostarting each procedure individually will conserve execution threads (each autostart routine gets its own thread).

TIP: To prevent autostart procedures from running when SQL Server first loads, start SQL Server with the 4022 trace flag. Adding –T4022 to the SQL Server command line tells the server not to run autostart procedures, but does not change their autostart status. The next time you start the server without the 4022 trace flag, they will again execute.

Executing a Stored Procedure via Remote Procedure Calls (RPC)

As an aside, one thing I should mention here is that the call to a stored procedure need not be a T-SQL batch. The ADO/OLEDB, ODBC, and DB-Library APIs all support executing stored procedures via RPC. Because it bypasses much of the usual statement and parameter processing, calling stored procedures via the RPC interface is more efficient than calling them via T-SQL batches. In particular, the RPC API facilitates the repetitive invocation of a routine with different sets of parameters. You can check this out in Query Analyzer (which uses the ODBC API) by changing the EXEC line in the script to the line in Listing 1–14:

Listing 1–14 Calling ListCustomersByCity via RPC

```
{CALL dbo.ListCustomersByCity}
```

This line uses the ODBC "call escape sequence" to invoke the routine using an RPC call. Restart the trace in Profiler, then execute the CALL command in Query Analyzer. You should see something like the following in the Profiler events window:
(Results abridged)

```
EventClass          TextData
------------------- ------------------------------
RPC:Starting        exec dbo.ListCustomersByCity
SP:ExecContextHit
SP:Starting         exec dbo.ListCustomersByCity
SP:Completed        exec dbo.ListCustomersByCity
RPC:Completed       exec dbo.ListCustomersByCity
```

Note the absence of the BatchStarting event. Instead, we have an RPC:Starting event followed, ultimately, by an RPC:Completed event. This tells us that the RPC API is being used to invoke the procedure. The procedure cache is unaffected by the switch to the RPC API; we still execute the procedure using the plan in the procedure cache.

Temporary Procedures

You create temporary procedures the same way you create temporary tables—a prefix of a single pound sign (**#**) creates a local temporary procedure that is

visible only to the current connection, whereas a double pound sign prefix (**##**) creates a global temporary procedure all connections can access.

Temporary procedures are useful when you want to combine the advantages of using stored procedures such as execution plan reuse and improved error handling with the advantages of ad hoc code. Because you can build and execute a temporary stored procedure at run-time, you get the best of both worlds. For the most part, sp_executesql can alleviate the necessity for temporary procedures, but they're still nice to have around when your needs exceed the capabilities of sp_executesql.

System Procedures

System procedures reside in the master database and are prefixed with **sp_**. You can execute a system procedure from any database. When executed from a database other than the master, a system procedure runs within the context of that database. So, for example, if the procedure references the sysobjects table (which exists in every database) it will access the one in the database that was current when it was executed, not the one in the master database, even though the procedure actually resides in the master. Listing 1–15 is a simple system procedure that lists the names and creation dates of the objects that match a mask:

Listing 1–15 A user-created system procedure that lists objects and their creation dates.

```
USE master
IF OBJECT_ID('dbo.sp_created') IS NOT NULL
  DROP PROC dbo.sp_created
GO
CREATE PROC dbo.sp_created @objname sysname=NULL
/*
Object: sp_created
Description: Lists the creation date(s) for the specified object(s)

Usage: sp_created @objname="Object name or mask you want to display"

Returns: (None)

$Author: Khen $. Email: khen@khen.com

$Revision: 2 $

Example: sp_created @objname="myprocs%"
```

```
Created: 1999-08-01. $Modtime: 1/04/01 12:16a $.
*/
AS
IF (@objname IS NULL) or (@objname='/?') GOTO Help
SELECT name, crdate FROM sysobjects
WHERE name like @objname
RETURN 0

Help:
EXEC dbo.sp_usage @objectname='sp_created',
       @desc='Lists the creation date(s) for the specified object(s)',
       @parameters='@objname="Object name or mask you want to display"',
       @example='sp_created @objname="myprocs%"',
       @author='Ken Henderson',
       @email='khen@khen.com',
       @version='1', @revision='0',
       @datecreated='19990801', @datelastchanged='19990815'
RETURN -1
GO
USE Northwind
EXEC dbo.sp_created 'Order%'
```

(Results)

```
name                    crdate
----------------------- -------------------------
Order Details           2000-08-06 01:34:08.470
Order Details Extended  2000-08-06 01:34:10.873
Order Subtotals         2000-08-06 01:34:11.093
Orders                  2000-08-06 01:34:06.610
Orders Qry              2000-08-06 01:34:09.780
```

As I've said, any system procedure, whether it's one you've created or one that ships with SQL Server, will use the current database context when executed. Listing 1–16 presents an example that uses one of SQL Server's own system stored procedures. It can be executed from any database to retrieve info on that database:

Listing 1–16 System procedures assume the current database context when executed.

```
USE Northwind
EXEC dbo.sp_spaceused
```

```
database_name                    database_size      unallocated space
-------------------------------- ------------------ ------------------
Northwind                        163.63 MB          25.92 MB

reserved           data               index_size         unused
------------------ ------------------ ------------------ ------------------
4944 KB            2592 KB            1808 KB            544 KB
```

Sp_spaceused queries several of SQL Server's system tables to create the report it returns. Because it's a system procedure, it automatically reflects the context of the current database even though it resides in the master database.

Note that, regardless of the current database, you can force a system procedure to run in the context of a given database by qualifying its name with the database name (as though it resided in that database) when you invoke it. Listing 1–17 illustrates:

Listing 1–17 You can force a system procedure to assume a specific database context.

```
USE pubs
EXEC Northwind..sp_spaceused
database_name                    database_size      unallocated space
-------------------------------- ------------------ ------------------
Northwind                        163.63 MB          25.92 MB

reserved           data               index_size         unused
------------------ ------------------ ------------------ ------------------
4944 KB            2592 KB            1808 KB            544 KB
```

In this example, even though sp_spaceused resides in the master and the current database is pubs, sp_spaceused reports space utilization info for the Northwind database because we qualified its name with Northwind when we invoked it. SQL Server correctly locates sp_spaceused in the master and executes it within the context of the Northwind database.

System Objects versus System Procedures

User-created system procedures are listed as user objects rather than system objects in Enterprise Manager. Why? Because the system bit of a procedure's

status column in sysobjects (0xC0000000) isn't set by default. You can call the undocumented procedure sp_MS_marksystemobject to set this bit. The lone parameter taken by the procedure is the name of the object with the system bit you wish to set. Many undocumented functions and DBCC command verbs do not work properly unless called from a system object (See Chapter 22 for more information). Check the IsMSShipped property of the OBJECTPROPERTY() function to determine whether an object's system bit has been set. Listing 1–18 is a code fragment that demonstrates this function:

Listing 1–18 System procedures and system objects are two different things.

```
USE master
GO
IF OBJECT_ID('dbo.sp_test') IS NOT NULL
  DROP PROC dbo.sp_test
GO
CREATE PROC dbo.sp_test AS
select 1
GO
SELECT OBJECTPROPERTY(OBJECT_ID('dbo.sp_test'),'IsMSShipped') AS 'System
Object?', status, status & 0xC0000000
FROM sysobjects WHERE NAME = 'sp_test'
GO
EXEC sp_MS_marksystemobject 'sp_test'
GO
SELECT OBJECTPROPERTY(OBJECT_ID('dbo.sp_test'),'IsMSShipped') AS 'System
Object?', status, status & 0xC0000000
FROM sysobjects WHERE NAME = 'sp_test'
```

(Results)

```
System Object? status
-------------- ----------- -----------
0              1610612737  1073741824

(1 row(s) affected)

System Object? status
-------------- ----------- -----------
1              -536870911  -1073741824
(1 row(s) affected)
```

As I've said, there are a variety of useful features that do not work correctly outside system procedures. For example, a stored procedure can't manipulate full text indexes via DBCC CALLFULLTEXT() unless its system bit is set. Regardless of whether you actually end up using this functionality, it's instructive to at least know how it works.

Extended Stored Procedures

Extended procedures are routines residing in DLLs that function similarly to regular stored procedures. They receive parameters and return results via SQL Server's Open Data Services API and are usually written in C or C++. They must reside in the master database and run within the SQL Server process space.

Although the two are similar, calls to extended procedures work a bit differently than calls to system procedures. Extended procedures aren't automatically located in the master database and they don't assume the context of the current database when executed. To execute an extended procedure from a database other than the master, you have to fully qualify the reference (e.g., EXEC master.dbo.xp_cmdshell 'dir').

A technique for working around these differences is to "wrap" an extended procedure in a system stored procedure. This allows it to be called from any database without requiring the **master** prefix. This technique is used with a number of SQL Server's own extended procedures. Many of them are wrapped in system stored procedures that have no purpose other than to make the extended procedures they call a bit handier. Listing 1–19 is an example of a system procedure wrapping a call to an extended procedure:

Listing 1–19 System procedures are commonly used to "wrap" extended procedures.

```
USE master
IF (OBJECT_ID('dbo.sp_hexstring') IS NOT NULL)
  DROP PROC dbo.sp_hexstring
GO
CREATE PROC dbo.sp_hexstring @int varchar(10)=NULL, @hexstring
varchar(30)=NULL OUT
/*
Object: sp_hexstring
Description: Return an integer as a hexadecimal string
```

```
Usage: sp_hexstring @int=Integer to convert, @hexstring=OUTPUT parm to
receive hex string

Returns: (None)

$Author: Khen $. Email: khen@khen.com

$Revision: 1 $

Example: sp_hexstring "23", @myhex OUT
Created: 1999-08-02. $Modtime: 1/4/01 8:23p $.
*/
AS
IF (@int IS NULL) OR (@int = '/?') GOTO Help
DECLARE @i int, @vb varbinary(30)
SELECT @i=CAST(@int as int), @vb=CAST(@i as varbinary)
EXEC master.dbo.xp_varbintohexstr @vb, @hexstring OUT
RETURN 0
Help:
EXEC sp_usage @objectname='sp_hexstring',
       @desc='Return an integer as a hexadecimal string',
       @parameters='@int=Integer to convert, @hexstring=OUTPUT parm to
receive hex string',
       @example='sp_hexstring "23", @myhex OUT',
       @author='Ken Henderson',
       @email='khen@khen.com',
       @version='1', @revision='0',
       @datecreated='19990802', @datelastchanged='19990815'
RETURN -1

GO
DECLARE @hex varchar(30)
EXEC sp_hexstring 10, @hex OUT
SELECT @hex
```

(Results)

```
------------------------------
0x0000000A
```

The whole purpose of sp_hexstring is to clean up the parameters to be passed to the extended procedure xp_varbintohexstr before calling it. Because sp_hexstring is a system procedure, it can be called from any database without requiring the caller to reference xp_varbintohexstr directly.

Internal Procedures

A number of system-supplied stored procedures are neither true system procedures nor extended procedures—they're implemented internally by SQL Server. Examples of these include sp_executesql, sp_xml_preparedocument, most of the sp_cursor routines, sp_reset_connection, and so forth. These routines have stubs in master..sysobjects, and are listed as extended procedures, but they are actually implemented internally by the server, not within an external ODS-based DLL. This is important to know because you cannot drop these or replace them with updated DLLs. They can be replaced only by patching SQL Server itself, which normally only happens when you apply a service pack.

Environmental Issues

A number of SQL Server environmental settings affect the behavior of stored procedures. You specify most of these via SET commands. They control the way that stored procedures handle nulls, quotes, cursors, BLOB fields, and so forth. Two of these—QUOTED_IDENTIFIER and ANSI_NULLS—are stored permanently in each procedure's status field in sysobjects, as I mentioned earlier in the chapter. That is, when you create a stored procedure, the status of these two settings is stored along with it. QUOTED_IDENTIFIER controls whether strings within double quotes are interpreted as object identifiers (e.g., table or column references), and ANSI_NULLS controls whether non-ANSI equality comparisons with NULLs are allowed.

SET QUOTED_IDENTIFIER is normally used with a stored procedure to allow the procedure to reference objects with names that contain reserved words, spaces, or other illegal characters. An example is provided in Listing 1–20.

Listing 1–20 SET QUOTED_IDENTIFIER allows references to objects with names with embedded spaces.

```
USE Northwind
SET QUOTED_IDENTIFIER ON
GO
```

```
IF OBJECT_ID('dbo.listorders') IS NOT NULL
  DROP PROC dbo.listorders
GO
CREATE PROC dbo.listorders
AS
SELECT * FROM "Order Details"
GO
SET QUOTED_IDENTIFIER OFF
GO
EXEC dbo.listorders
```

(Results abridged)

```
OrderID     ProductID   UnitPrice             Quantity Discount
----------- ----------- --------------------- -------- --------------------
10248       11          14.0000               12       0.0
10248       42          9.8000                10       0.0
10248       72          34.8000               5        0.0
10249       14          18.6000               9        0.0
10249       51          42.4000               40       0.0
10250       41          7.7000                10       0.0
```

"Order Details" contains both a reserved word and a space, so it can't be referenced without special handling. In this case, we turned on quoted identifier support and enclosed the table name in double quotes, but a better way would be to use SQL Server's square brackets ([ ]) to enclose the name (e.g., [Order Details]) because this alleviates the need to change any settings. Note that bracketed object names are not supported by the ANSI/ISO SQL standard.

The ANSI_NULLS setting is even more useful to stored procedures. It controls whether non-ANSI equality comparisons with NULLs work properly. This is particularly important with stored procedure parameters that can receive NULL values. See Listing 1–21 for an example:

Listing 1–21 SET ANSI_NULLS allows comparisons between variables or columns and NULL values to work as you would expect.

```
USE Northwind
IF (OBJECT_ID('dbo.ListRegionalEmployees') IS NOT NULL)
  DROP PROC dbo.ListRegionalEmployees
GO
SET ANSI_NULLS OFF
GO
CREATE PROC dbo.ListRegionalEmployees @region nvarchar(30)
```

```
AS
SELECT EmployeeID, LastName, FirstName, Region FROM employees
WHERE Region=@region

GO
SET ANSI_NULLS ON
GO

EXEC dbo.ListRegionalEmployees NULL
```

(Results)

```
EmployeeID  LastName             FirstName  Region
----------- -------------------- ---------- ---------------
5           Buchanan             Steven     NULL
6           Suyama               Michael    NULL
7           King                 Robert     NULL
9           Dodsworth            Anne       NULL
```

Thanks to SET ANSI_NULLS, the procedure can successfully compare a NULL @region with the **region** column in the Northwind Employees table. The query returns the rows that have NULL region values because, contrary to the ANSI SQL specification, SQL Server checks the NULL variable against the column for equality. The handiness of this becomes more evident when a procedure defines a large number of "NULL-able" parameters. Without the ability to test NULL values for equality in a manner identical to non-NULL values, each NULL-able parameter would require special handling (perhaps using the IS NULL predicate), very likely multiplying the amount of code necessary to process query parameters.

Because SQL Server stores the QUOTED_IDENTIFIER and ANSI_NULLS settings with each stored procedure, you can trust them to have the values you require when a procedure runs. The server restores them to the values they had when the procedure was created each time the procedure runs, then resets them afterward. Here's an example:

```
SET ANSI_NULLS ON
EXEC dbo.ListRegionalEmployees NULL
```

The stored procedure still executes as though ANSI_NULLS is set to OFF. Note that you can check the saved status of a procedure's QUOTED_IDENTIFIER and ANSI_NULLS settings via the OBJECTPROPERTY() function. An example is provided in Listing 1–22:

Listing 1–22 You can check the ANSI_NULLS and QUOTED_IDENTIFIER status for a procedure using the OBJECTPROPERTY function.

```
USE Northwind
SELECT OBJECTPROPERTY(OBJECT_ID('dbo.ListRegionalEmployees'),
'ExecIsAnsiNullsOn') AS 'AnsiNulls'
```

(Results)

```
AnsiNulls
-----------
0
```

A number of other environmental commands affect how stored procedures execute. SET XACT_ABORT, SET CURSOR_CLOSE_ON_COMMIT, SET TEXTSIZE, SET IMPLICIT_TRANSACTIONS, and numerous others help determine how a stored procedure behaves when executed. If you have a stored procedure that requires a SET command to have a particular value to run properly, set it to that value as early as possible in the procedure and document why it's necessary via comments.

Parameters

Parameters can be passed to stored procedures by name or by position. An example of each method is presented in Listing 1–23:

Listing 1–23 You can pass procedure parameters by position or by name.

```
EXEC dbo.sp_who 'sa'
EXEC dbo.sp_who @loginame='sa'
```

Obviously, the advantage of referencing parameters by name is that you can specify them out of order.

You can force a parameter for which a default value has been defined to use that default by omitting it altogether or by passing it the DEFAULT keyword, as in Listing 1–24:

Listing 1–24 Passing DEFAULT for a parameter causes it to assume its default value.

```
EXEC dbo.sp_who @loginame=DEFAULT
```

You can specify NULL to supply individual parameters with NULL values. This is sometimes handy for procedures that expose special features when parameters are omitted or set to NULL. An example is presented in Listing 1–25:

Listing 1–25 You can pass NULL to a parameter.

```
EXEC dbo.sp_who @loginame=NULL
```

(Results abridged)

```
spid   ecid   status                          loginame
------ ------ ------------------------------ -----------
1      0      background                      sa
2      0      background                      sa
3      0      sleeping                        sa
4      0      background                      sa
5      0      background                      sa
6      0      sleeping                        sa
7      0      background                      sa
8      0      background                      sa
9      0      background                      sa
10     0      background                      sa
11     0      background                      sa
12     0      background                      sa
13     0      background                      sa
51     0      sleeping                        SKREWYTHIN\khen
52     0      sleeping                        SKREWYTHIN\khen
53     0      sleeping                        SKREWYTHIN\khen
```

Here, sp_who returns a list of all active connections because its @loginame parameter is passed NULL. When a valid login name is specified, sp_who returns only those connections established by the specified login name. You'd see the same result if @loginame had not been supplied at all—all connections would be listed.

Return Status Codes

Procedures return status codes via the RETURN command. For an example, see Listing 1–26:

Listing 1–26 Use RETURN to render stored procedure status codes.

```
RETURN(-100)
-- and
RETURN -100
```

These return a status code of –100 to the caller of the procedure (the parameters are optional). A return code of 0 indicates success, values –1 through –14 indicate different types of failures (see the Books Online for descriptions of these), and values –15 through –99 are reserved for future use.

You can access a procedure's return code by assigning it to an integer variable, as in Listing 1–27:

Listing 1–27 You can save a procedure's return status code to an integer variable.

```
DECLARE @res int
EXEC @res=dbo.sp_who
SELECT @res
```

Output Parameters

In addition to the return status code that every stored procedure supports, you can use output parameters to return other types of values from a procedure. These parameters can be integers, character strings, dates, and even cursors. An example is provided in Listing 1–28:

Listing 1–28 Cursor output parameters are handy for returning result sets.

```
USE pubs
IF OBJECT_ID('dbo.listsales') IS NOT NULL
  DROP PROC dbo.listsales
GO
CREATE PROC dbo.listsales @bestseller tid OUT, @topsales int OUT,
                      @salescursor cursor varying OUT
AS
SELECT @bestseller=bestseller, @topsales=totalsales
FROM (
      SELECT TOP 1 title_id AS bestseller, SUM(qty) AS totalsales
      FROM sales
      GROUP BY title_id
```

```
      ORDER BY 2 DESC) bestsellers

DECLARE s CURSOR
LOCAL
FOR SELECT * FROM sales

OPEN s

SET @salescursor=s
RETURN(0)
GO

DECLARE @topsales int, @bestseller tid, @salescursor cursor
EXEC dbo.listsales @bestseller OUT, @topsales OUT, @salescursor OUT
SELECT @bestseller, @topsales
FETCH @salescursor
CLOSE @salescursor
DEALLOCATE @salescursor
```

(Results abridged)

```
------ -----------
PS2091 108

stor_id ord_num   ord_date   qty  payterms  title_id
------- --------- ---------- ---- --------- --------
6380    6871      1994-09-14 5    Net 60    BU1032
```

Using a cursor output parameter is a good alternative for returning a result set to a caller. By using a cursor output parameter rather than a traditional result set, you give the caller control over how and when to process the result set. The caller can also determine various details about the cursor through system function calls before actually processing the result.

Output parameters are identified with the OUTPUT keyword (you can abbreviate this as "OUT"). Note the use of the OUT keyword in the procedure definition as well as in the EXEC parameter list. Output parameters must be identified in a procedure's parameter list as well as when the procedure is called.

The VARYING keyword is required for cursor parameters and indicates that the return value is nonscalar—that is, it can return more than one value. Cursor parameters can be output parameters only, so the OUT keyword is also required.

Listing Procedure Parameters

You can list a procedure's parameters (which include its return status code, considered parameter 0) by querying the INFORMATION_SCHEMA.PARAMETERS view (Listing 1–29).

Listing 1–29 INFORMATION_SCHEMA.PARAMETERS returns stored procedure parameter info.

```
USE Northwind
SELECT PARAMETER_MODE, PARAMETER_NAME, DATA_TYPE
FROM INFORMATION_SCHEMA.PARAMETERS
WHERE SPECIFIC_NAME='Employee Sales by Country'
```

(Results abridged)

```
PARAMETER_MODE PARAMETER_NAME   DATA_TYPE
-------------- ---------------- ----------
IN             @Beginning_Date  datetime
IN             @Ending_Date     datetime
```

General Parameter Notes

In addition to what I've already said about parameters, here are a few more tips:

- Check stored procedure parameters for invalid values early on.
- Human-friendly names allow parameters to be passed by name more easily.
- It's a good idea to provide default values for parameters when you can. This makes a procedure easier to use. A parameter default can consist of a constant or the NULL value.
- Because parameter names are local to stored procedures, you can use the same name in multiple procedures. If you have ten procedures that each take a user name parameter, name the parameter @UserName in all ten of them—for simplicity's sake and for general consistency in your code.
- Procedure parameter information is stored in the syscolumns system table.
- A stored procedure can receive as many as 1,024 parameters. If you have a procedure that you think needs more parameters than 1,024, you should probably consider redesigning it.
- The number and size of stored procedure local variables is limited only by the amount of memory available to SQL Server.

Table 1–1 Stored Procedure-Related Functions

Function	Returns
@@FETCH_STATUS	The status of the last FETCH operation
@@NESTLEVEL	The current procedure nesting level
@@OPTIONS	A bitmap of the currently specified user options
@@PROCID	The object ID of the current procedure
@@SPID	The process ID of the current process
@@TRANCOUNT	The current transaction nesting level

Automatic Variables, a.k.a. System Functions

By their very nature, automatic variables, also known as *system functions*, are usually the province of stored procedures. This makes most of them germane in some way to a discussion about stored procedures. Several, in fact, are used almost exclusively in stored procedures. Table 1–1 summarizes them.

Flow Control Language

Certain Transact-SQL commands affect the order in which statements are executed in a stored procedure or command batch. These are referred to as *flow control* or *control-of-flow statements* because they control the flow of Transact-SQL code execution. Transact-SQL flow control language statements include IF…ELSE, WHILE, GOTO, RETURN, WAITFOR, BREAK, CONTINUE, and BEGIN…END. We'll discuss the various flow control commands further in the book, but for now here's a simple procedure that illustrates all of them (Listing 1–30):

Listing 1–30 Flow control statements as they behave in the wild.

```
USE pubs
IF OBJECT_ID('dbo.listsales') IS NOT NULL
  DROP PROC dbo.listsales
GO
CREATE PROC dbo.listsales @title_id tid=NULL
AS

IF (@title_id='/?') GOTO Help       -- Here's a basic IF
```

```
-- Here's one with a BEGIN..END block
IF NOT EXISTS(SELECT * FROM titles WHERE title_id=@title_id) BEGIN
  PRINT 'Invalid title_id'
  WAITFOR DELAY '00:00:03' -- Delay 3 secs to view message
  RETURN -1
END

IF NOT EXISTS(SELECT * FROM sales WHERE title_id=@title_id) BEGIN
  PRINT 'No sales for this title'
  WAITFOR DELAY '00:00:03' -- Delay 3 secs to view message
  RETURN -2
END

DECLARE @qty int, @totalsales int
SET @totalsales=0

DECLARE c CURSOR
FOR SELECT qty FROM sales WHERE title_id=@title_id
OPEN c

FETCH c INTO @qty

WHILE (@@FETCH_STATUS=0) BEGIN      -- Here's a WHILE loop
  IF (@qty<0) BEGIN
    Print 'Bad quantity encountered'
    BREAK    -- Exit the loop immediately
  END ELSE IF (@qty IS NULL) BEGIN
    Print 'NULL quantity encountered -- skipping'
    FETCH c INTO @qty
    CONTINUE -- Continue with the next iteration of the loop
  END
  SET @totalsales=@totalsales+@qty
  FETCH c INTO @qty
END

CLOSE c
DEALLOCATE c

SELECT @title_id AS 'TitleID', @totalsales AS 'TotalSales'
RETURN 0     -- Return from the procedure indicating success

Help:
EXEC sp_usage @objectname='listsales',
       @desc='Lists the total sales for a title',
```

```
        @parameters='@title_id="ID of the title you want to check"',
        @example='EXEC listsales "PS2091"',
        @author='Ken Henderson',
        @email='khen@khen.com',
        @version='1', @revision='0',
        @datecreated='19990803', @datelastchanged='19990818'
WAITFOR DELAY '00:00:03' -- Delay 3 secs to view message
RETURN -1
GO

EXEC dbo.listsales 'PS2091'
EXEC dbo.listsales 'badone'
EXEC dbo.listsales 'PC9999'

TitleID TotalSales
------- -----------
PS2091  191
Invalid title_id
No sales for this title
```

Errors

The @@ERROR automatic variable returns the error code of the last Transact-SQL statement. If there was no error, @@ERROR returns zero. Because @@ERROR is reset after each Transact-SQL statement, you must save it to a variable if you wish to process it further after checking it.

If you want to write robust code that runs for years without having to be reengineered, make a habit of checking @@ERROR often in your stored procedures, especially after data modification statements. A good indicator of resilient code is consistent error checking, and until Transact-SQL supports structured exception handling, checking @@ERROR frequently is the best way to protect your code against unforeseen circumstances.

Error Messages

The system procedure sp_addmessage adds custom messages to the sysmessages table that can then be raised (returned to the client) by the RAISERROR command. User messages should have error numbers of 50,000 or higher. The chief advantage of using SQL Server's system messages facility is internationalization. Because you specify a language ID when you add a message via

sp_addmessage, you can add a separate version of your application's messages for each language it supports. When your stored procedures then reference a message by number, the appropriate message will be returned to your application using SQL Server's current language setting.

RAISERROR

Stored procedures report errors to client applications via the RAISERROR command. RAISERROR doesn't change the flow of a procedure; it merely displays an error message, sets the @@ERROR automatic variable, and optionally writes the message to the SQL Server error log and the NT application event log. RAISERROR can reference an error message added to the sysmessages table via the sp_addmessage system procedure, or you can supply it a message string of your own. If you pass a custom message string to RAISERROR, the error number is set to 50,000; if you raise an error by number using a message ID in the sysmessages table, @@ERROR is assigned the message number you raise. RAISERROR can format messages similarly to the C PRINTF() function, allowing you to supply your own arguments for the error messages it returns.

Both a severity and a state can be specified when raising an error message with RAISERROR. Severity values less than 16 produce informational messages in the application event log (when logged). A severity of 16 produces a warning message in the event log. Severity values greater than 16 produce error messages in the event log. Severity values up through 18 can be raised by any user; severity values 19 through 25 are reserved for members of the sysadmin role and require the use of the WITH LOG option. Severity values of 20 and higher are considered fatal and cause the client connection to be terminated.

State has no predefined meaning to SQL Server; it's an informational value that you can use to return state information to an application. Raising an error with a state of 127 will cause the ISQL and OSQL utilities to set the operating system ERRORLEVEL variable to the error number returned by RAISERROR.

The WITH LOG option copies the error message to the NT event log (if SQL Server is running on Windows NT, Windows 2000, or Windows XP) and the SQL Server error log regardless of whether the message was defined using the WITH_LOG option of sp_addmessage. The WITH NOWAIT option causes the message to be returned immediately to the client. The WITH SETERROR option forces @@ERROR to return the last error number raised, regardless of the severity of the error message. See Chapter 7 for detailed examples of how to use RAISERROR(), @@ERROR, and SQL Server's other error-handling mechanisms.

Nesting

You can nest stored procedure calls up to 32 levels deep. Use the @@NESTLEVEL automatic variable to check the nesting level from within a stored procedure or trigger. From a command batch, @@NESTLEVEL returns 0. From a stored procedure called from a command batch and from first-level triggers, @@NESTLEVEL returns 1. From a procedure or trigger called from nesting level 1, @@NESTLEVEL returns 2; procedures called from level 2 procedures return level 3, and so on. Objects (including temporary tables) and cursors created within a stored procedure are visible to all objects it calls. Objects and cursors created in a command batch are visible to all the objects referenced in the command batch.

Recursion

Because Transact-SQL supports recursion, you can write stored procedures that call themselves. Recursion can be defined as a method of problem solving wherein the solution is arrived at by repetitively applying it to subsets of the problem. A common application of recursive logic is to perform numeric computations that lend themselves to repetitive evaluation by the same processing steps. Listing 1–31 presents an example that features a stored procedure that calculates the factorial of a number:

Listing 1–31 Stored procedures can call themselves recursively.

```
SET NOCOUNT ON
USE master
IF OBJECT_ID('dbo.sp_calcfactorial') IS NOT NULL
  DROP PROC dbo.sp_calcfactorial
GO
CREATE PROC dbo.sp_calcfactorial @base_number decimal(38,0), @factorial
decimal(38,0) OUT
AS
SET NOCOUNT ON
DECLARE @previous_number decimal(38,0)

IF ((@base_number>26) and (@@MAX_PRECISION<38)) OR (@base_number>32) BEGIN
  RAISERROR('Computing this factorial would exceed the server''s max.
numeric precision of %d or the max. procedure nesting level of
32',16,10,@@MAX_PRECISION)
```

```
  RETURN(-1)
END

IF (@base_number<0) BEGIN
  RAISERROR('Can''t calculate negative factorials',16,10)
  RETURN(-1)
END

IF (@base_number<2) SET @factorial=1 -- Factorial of 0 or 1=1
ELSE BEGIN
  SET @previous_number=@base_number-1
  EXEC dbo.sp_calcfactorial @previous_number, @factorial OUT -- Recursive
call
  IF (@factorial=-1) RETURN(-1) -- Got an error, return
  SET @factorial=@factorial*@base_number
  IF (@@ERROR<>0) RETURN(-1) -- Got an error, return
END
RETURN(0)
GO

DECLARE @factorial decimal(38,0)
EXEC dbo.sp_calcfactorial 32, @factorial OUT
SELECT @factorial
```

The procedure begins by checking to make sure it has been passed a valid number for which to compute a factorial. It then recursively calls itself to perform the computation. With the default maximum numeric precision of 38, SQL Server can handle numbers in excess of 263 decillion. (*Decillion* is the U.S. term for 1 followed by 33 zeros. In Great Britain, France, and Germany, 1 followed by 33 zeros is referred to as 1,000 quintillion.) As you'll see in Chapter 11, UDFs functions are ideal for computations like factorials.

Summary

In this chapter you learned the basics of writing stored procedures. We discussed how to monitor stored procedure activity using the Profiler utility, and we dealt with several real-world stored procedure programming issues. You learned about the procedure cache, how SQL Server uses it, and how you can watch it for signs of inefficiencies in your code. You learned about many of the nuances and quirks in SQL Server's stored procedure programming language,

Transact-SQL, and you learned how to use them to your advantage and/or how to work around them as appropriate. You learned how to pass parameters to stored procedures, how to return stored procedure status codes, and how to return data via output parameters. We talked about how to nest stored procedures, as well as how to call them recursively. Hopefully, through all this, you've begun to glimpse a bit of the power available to you in Transact-SQL and SQL Server stored procedures. We'll build on this throughout the remainder of the book.