Inside Microsoft® SQL Server™ 2005: T-SQL Programming

Itzik Ben-Gan (Solid Quality Learning), Dejan Sarka; Roger Wolter

To learn more about this book, visit Microsoft Learning at http://www.microsoft.com/MSPress/books/8564.aspx

9780735621978
Publication Date: May 2006

Microsoft Press

Contents

Chapter 3

Cursors

In this chapter:

You probably won't find many database professionals arguing about the necessity for SELECT statements, but many argue about whether cursors are necessary. Arguments also arise about the use of temporary tables; dynamic code; and integrated XML, XQuery, and CLR. There must be a reason why database professionals are in complete agreement about some aspects of Microsoft SQL Server 2005 but have conflicting opinions about other aspects (call the constructs under conflicting opinions *arguable constructs*). Let me offer my two cents' worth on the subject.

These arguable constructs have a high potential for misuse because database professionals often lack knowledge and experience in set-based querying and the relational model. Such misuse can lead to very poor implementations. Defenders of these arguable constructs would argue that any construct can be abused because of lack of knowledge and experience. Still, I think that there is a difference between these constructs and many others. Knives and matches are very useful tools, but only in the hands of responsible people. You wouldn't want those devices in the hands of children. Even with no bad intentions on the part of the users, the potential for catastrophe is high. A child could also do damage with crayons and books, but the likelihood of that happening is much lower and the damage wouldn't be as severe.

I didn't say that I side with those who oppose the arguable constructs, or that I'm on any side for that matter. But I do think that placing such tools in the hands of programmers who lack adequate knowledge of set-based querying and the relational model can yield bad results. The key is having the maturity to recognize the appropriate time and place to use each construct (static set-based queries, dynamic SQL, cursors, XML, CLR, table expressions, temporary tables, and so on). This book tries to guide you to that level of maturity.

I hope you will forgive me for the philosophical approach to this subject, but for me SQL is a "way" that has important philosophical aspects. In my mind—and you don't have to agree—I separate the careers of T-SQL programmers into three typical phases:

1. **Procedural.** This is the phase in which programmers have just started to work with databases. They have insufficient experience working with the relational model

and set-based thinking. In this phase, it's common to see misuse of tools such as cursors, temporary tables, dynamic execution, and procedural coding in general. Programmers at this stage are usually oblivious to the damage that they're causing.

2. **Becoming sober.** This is the phase in which programmers realize there's more to database programming—that SQL is not a nuisance that interferes with writing procedural code but, rather, it's based on the strong foundations of set theory and the relational model. In this phase, programmers tend to believe "experts" who say cursors, temporary tables, and dynamic execution are evil and should never be used. At this point, programmers either avoid using such constructs altogether or really feel bad about the code they write. There's usually lack of confidence at this stage.
3. **Maturity.** This stage is characterized by the void or Zen mindset. In this phase, programmers have deep knowledge and understanding, and they feel confident about their code. This doesn't mean they stop pursuing deeper knowledge or improving fundamental techniques. In this phase, programmers apply set-based thinking for the most part, but they realize that there's a time and place for other constructs as well. I refer to this phase as the "void" in the positive and abstract sense—that is, programmers develop intuition regarding the type of solution that would fit a given task and don't need to spend much time determining which technique is appropriate.

Developing the intuition described in phase three involves knowing when the typical approach of using pure static SQL programming will not do the job. Although pure static SQL programming is typically the way to go, it will only get you so far in some cases. There are cases where using temporary tables can substantially improve performance; where dynamic execution actually overcomes complex problems; where the use of procedural languages such as C# and Visual Basic allows more flexibility without conflicting with the relational model; and where storing states of data in XML format makes sense. This book explores these cases in dedicated chapters and sections.

Using Cursors

In this chapter, I'll explain the types of problems for which cursors are a reasonable solution, even though such cases are not common. The goal of the chapter is to show you how to use them wisely.

I'll assume that you have sufficient technical knowledge of the various cursor types and know the syntax for declaring and using them. If you don't, you can find a lot of information about cursors in Books Online. My focus is to explain why cursors are typically not the right choice and to present the cases in which cursors do make sense.

So why should you avoid using cursors for the most part?

For one, cursors conflict with the main premise of the relational model. Using cursors, you apply procedural logic rather than set-based logic. That is, you write a lot of code with iterations, where you mainly focus on "how" to deal with data. When you apply set-based logic,

you typically write substantially less code, as you focus on "what" you want and not how to get it. You need to be able to recognize the cases where a problem is procedural/iterative in nature—where you truly need to process one row at a time. In these cases, you should consider using a cursor. For example, you have a table that contains user information along with e-mail addresses, and you need to send e-mail to all users. Or you need to invoke a stored procedure per each row in some table and provide the stored procedure with column values from each row as arguments.

Cursors also have a lot of overhead involved with the row-by-row manipulation and are typically substantially slower than set-based code (queries). I demonstrate the use of set-based solutions throughout the book. You need to be able to measure and estimate the cursor overhead and identify the few scenarios where cursors will yield better performance than set-based code. In some cases, data distribution will determine whether a cursor or a set-based solution will yield better performance.

There's another very important aspect of cursors—they can request and assume ordered data as input, whereas queries can accept only a relational input, which by definition cannot assume a particular order. This difference is important in identifying scenarios in which cursors might actually be faster—such as problems that are tightly based on ordered access to the data. Examples of such problems are running aggregations and ranking calculations, resolving some temporal problems, and so on. The I/O cost involved with the cursor activity plus the cursor overhead might end up being lower than a set-based solution that performs substantially more I/O.

ANSI recognizes the practical need for manipulation of ordered data and provides some standards for addressing this need. In extensions to the ANSI SQL:1999 standard and in the ANSI SQL:2003 standard, you can find several query constructs that inherently rely on ordering—for example, the ANSI OVER(ORDER BY ...) clause, which determines the calculation order for ranking and aggregate calculations, or the SEARCH clause defined with recursive CTEs, which determines the order of traversal of trees.

SQL Server 2005 implements the OVER clause with support for ORDER BY only for ranking functions, and SQL Server 2005's engine was, of course, enhanced to support the rapid performance of such calculations. As a result, ranking calculations using queries are now substantially faster than cursor-based solutions. With aggregate functions in SQL Server 2005, however, the OVER clause does not support ORDER BY. Therefore, set-based solutions to compute running aggregations with large groups of data are slower than cursor-based solutions. I'll demonstrate this in the section Running Aggregations later in this chapter. The SEARCH clause for recursive common table expressions (CTEs) has not been implemented in SQL Server 2005.

Another kind of problem where cursor solutions are faster than query solutions is matching problems, which I'll also demonstrate. With those, I haven't found set-based solutions that perform nearly as well as cursor solutions. But I haven't given up. One of my goals is to find set-based solutions for problems that are not procedural. Some of those problems could have

set-based solutions if newer ANSI constructs had been supported in SQL Server. I hope that SQL Server will implement those in future versions. And when the ANSI standard doesn't have answers, I believe there will be vendor-specific product extensions, followed by motions to the ANSI committee to add them to the standard.

Cursor Overhead

In this chapter's introduction, I talked about the benefits that set-based solutions have over cursor-based ones. I mentioned both logical and performance benefits. For the most part, efficiently written set-based solutions will outperform cursor-based solutions for two reasons.

First, you empower the optimizer to do what it's so good at—generating multiple valid execution plans, and choosing the most efficient one. When you apply a cursor-based solution, you're basically forcing the optimizer to go with a rigid plan that doesn't leave much room for optimization—at least not as much room as with set-based solutions.

Second, row-by-row manipulation creates a lot of overhead. You can run some simple tests to witness and measure this overhead—for example, just scanning a table with a simple query and comparing the results to scanning it with a cursor. To compare apples to apples, make sure you're scanning the same amount of data as you did with the cursor-based query. You can eliminate the actual disk I/O cost by running the code twice. (The first run will load the data to cache.) To eliminate the time it takes to generate the output, you should run your code with the Discard Results After Execution option in SQL Server Management Studio (SSMS) turned on. The difference in performance between the set-based code and the cursor code will then be the cursor's overhead.

I will now demonstrate how to compare scanning the same amount of data with set-based code versus with a cursor. Run the following code to generate a table called T1, with a million rows, each containing slightly more than 200 bytes:

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
GO
SELECT n AS keycol, CAST('a' AS CHAR(200)) AS filler
INTO dbo.T1
FROM dbo.Nums;

CREATE UNIQUE CLUSTERED INDEX idx_keycol ON dbo.T1(keycol);
```

You can find the code to create and populate the Nums table in Chapter 1.

Turn on the Discard Results After Execution option in SSMS (under Tools|Options|Query Results|SQL Server|Results to Grid or Results to Text). Now clear the cache:

```
DBCC DROPCLEANBUFFERS;
```

Run the following set-based code twice—the first run will measure performance against cold cache, and the second will measure it against warm cache:

```
SELECT keycol, filler FROM dbo.T1;
```

On my system, this query ran for 4 seconds against cold cache and 2 seconds against warm cache. Clear the cache again, and then run the cursor code twice:

```
DECLARE @keycol AS INT, @filler AS CHAR(200);
DECLARE C CURSOR FAST_FORWARD FOR SELECT keycol, filler FROM dbo.T1;
OPEN C
FETCH NEXT FROM C INTO @keycol, @filler;
WHILE @@fetch_status = 0
BEGIN
  -- Process data here
  FETCH NEXT FROM C INTO @keycol, @filler;
END
CLOSE C;
DEALLOCATE C;
```

This code ran for 22 seconds against cold cache and 20 seconds against warm cache. Considering the warm cache example, in which there's no physical I/O involved, the cursor code ran ten times more slowly than the set-based code, and notice that I used the fastest cursor you can get—FAST_FORWARD. Both solutions scanned the same amount of data. Besides the performance overhead, you also have the development and maintenance overhead of your code. This is a very basic example involving little code; in production environments with more complex code, the problem is, of course, much worse.

Dealing with Each Row Individually

Remember that cursors can be useful when the problem is a procedural one, and you must deal with each row individually. I provided examples of such scenarios earlier. Here I want to show an alternative to cursors that programmers may use to apply iterative logic, and compare its performance with the cursor code I just demonstrated in the previous section. Remember that the cursor code that scanned a million rows took approximately 20 seconds to complete. Another common technique to iterate through a table's rows is to loop through the keys and use a set-based query for each row. To test the performance of such a solution, make sure the Discard Results After Execution option in SSMS is still turned on. Then run the following code:

```
DECLARE @keycol AS INT, @filler AS CHAR(200);

SELECT @keycol = keycol, @filler = filler
FROM (SELECT TOP (1) keycol, filler
      FROM dbo.T1
      ORDER BY keycol) AS D;

WHILE @@rowcount = 1
BEGIN
  -- Process data here
```

```
  -- Get next row
  SELECT @keycol = keycol, @filler = filler
  FROM (SELECT TOP (1) keycol, filler
        FROM dbo.T1
        WHERE keycol > @keycol
        ORDER BY keycol) AS D;
END
```

This implementation is a bit "cleaner" than dealing with a cursor, and that's the aspect of it that I like. You use a TOP (1) query to grab the first row (based on key order). Within a loop, when a row was found in the previous iteration, you process the data and request the next row (the row with the next key). This code ran for about 90 seconds—several times slower than the cursor code. I created a clustered index on *keycol* to improve performance by accessing the desired row in each iteration with minimal I/O. Without that index, this code would run substantially slower because each invocation of the query would need to rescan large portions of data. A cursor solution based on sorted data would also benefit from an index and would run substantially slower without one because it would need to sort the data after scanning it. With large tables and no index on the sort columns, the sort operation can be very expensive because sorting in terms of complexity is $O(n \log n)$, while scanning is only $O(n)$.

Before you proceed, make sure you turn off the "Discard Results After Execution" option in SSMS.

Order-Based Access

In the introduction, I mentioned that cursors have the potential to yield better performance than set-based code when the problem is inherently order based. In this section, I'll show some examples. Where relevant, I'll discuss query constructs that ANSI introduces to allow for "cleaner" code that performs well without the use of cursors. However, some of these ANSI constructs have not been implemented in SQL Server 2005.

Custom Aggregates

In *Inside T-SQL Querying*, I discussed custom aggregates by describing problems that require you to aggregate data even though SQL Server doesn't provide such aggregates as built-in functions—for example, product of elements, string concatenation, and so on. I described four classes of solutions and demonstrated three of them: pivoting, which is limited to a small number of elements in a group; user-defined aggregates (UDAs) written in a .NET language, which force you to write in a language other than T-SQL and enable CLR support in SQL Server; and specialized solutions, which can be very fast but are applicable to specific cases and are not suited to generic use. Another approach to solving custom aggregate problems is using cursors. This approach is not very fast; nevertheless, it is straightforward, generic, and not limited to situations in which you have a small number of elements in a group. To see a demonstration of a cursor-based solution for custom aggregates, run the code in Listing 3-1 to

create and populate the Groups table, which I also used in my examples in *Inside T-SQL Querying*.

Listing 3-1 Creating and populating the Groups table

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Groups') IS NOT NULL
  DROP TABLE dbo.Groups;
GO

CREATE TABLE dbo.Groups
(
  groupid  VARCHAR(10) NOT NULL,
  memberid INT         NOT NULL,
  string   VARCHAR(10) NOT NULL,
  val      INT         NOT NULL,
  PRIMARY KEY (groupid, memberid)
);

INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('a', 3, 'stra1', 6);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('a', 9, 'stra2', 7);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 2, 'strb1', 3);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 4, 'strb2', 7);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 5, 'strb3', 3);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('b', 9, 'strb4', 11);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 3, 'strc1', 8);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 7, 'strc2', 10);
INSERT INTO dbo.Groups(groupid, memberid, string, val)
  VALUES('c', 9, 'strc3', 12);
```

Listing 3-2 shows cursor code that calculates the aggregate product of the *val* column for each group represented by the *groupid* column, and it generates the output shown in Table 3-1.

Listing 3-2 Cursor code for custom aggregate

```
DECLARE
  @Result TABLE(groupid VARCHAR(10), product BIGINT);
DECLARE
  @groupid AS VARCHAR(10), @prvgroupid AS VARCHAR(10),
  @val AS INT, @product AS BIGINT;

DECLARE C CURSOR FAST_FORWARD FOR
  SELECT groupid, val FROM dbo.Groups ORDER BY groupid;
```

```
OPEN C

FETCH NEXT FROM C INTO @groupid, @val;
SELECT @prvgroupid = @groupid, @product = 1;

WHILE @@fetch_status = 0
BEGIN
  IF @groupid <> @prvgroupid
  BEGIN
    INSERT INTO @Result VALUES(@prvgroupid, @product);
    SELECT @prvgroupid = @groupid, @product = 1;
  END

  SET @product = @product * @val;

  FETCH NEXT FROM C INTO @groupid, @val;
END

IF @prvgroupid IS NOT NULL
  INSERT INTO @Result VALUES(@prvgroupid, @product);

CLOSE C;

DEALLOCATE C;

SELECT groupid, product FROM @Result;
```

Table 3-1 Aggregate Product

Groupid	*product*
A	42
B	693
C	960

The algorithm is straightforward: scan the data in *groupid* order; while traversing the rows in the group, keep multiplying by *val*; and whenever the *groupid* value changes, store the result of the product for the previous group aside in a table variable. When the loop exits, you still hold the aggregate product for the last group, so store it in the table variable as well unless the input was empty. Finally, return the aggregate products of all groups as output.

Running Aggregations

The previous problem, which discussed custom aggregates, used a cursor-based solution that scanned the data only once, but so did the pivoting solution, the UDA solution, and some of the specialized set-based solutions. If you consider that cursors incur more overhead than set-based solutions that scan the same amount of data, you can see that the cursor-based solutions are bound to be slower. On the other hand, set-based solutions for running aggregation problems in SQL Server 2005 involve rescanning portions of the data multiple times, whereas the cursor-based solutions scan the data only once.

I covered Running Aggregations in *Inside T-SQL Querying*. Here, I'll demonstrate cursor-based solutions. Run the following code, which creates and populates the EmpOrders table:

```
USE tempdb;
GO

IF OBJECT_ID('dbo.EmpOrders') IS NOT NULL
  DROP TABLE dbo.EmpOrders;
GO

CREATE TABLE dbo.EmpOrders
(
  empid    INT      NOT NULL,
  ordmonth DATETIME NOT NULL,
  qty      INT      NOT NULL,
  PRIMARY KEY(empid, ordmonth)
);

INSERT INTO dbo.EmpOrders(empid, ordmonth, qty)
  SELECT O.EmployeeID,
    CAST(CONVERT(CHAR(6), O.OrderDate, 112) + '01'
      AS DATETIME) AS ordmonth,
    SUM(Quantity) AS qty
  FROM Northwind.dbo.Orders AS O
    JOIN Northwind.dbo.[Order Details] AS OD
      ON O.OrderID = OD.OrderID
  GROUP BY EmployeeID,
    CAST(CONVERT(CHAR(6), O.OrderDate, 112) + '01'
      AS DATETIME);
```

This is the same table and sample data I used in *Inside T-SQL Querying* to demonstrate set-based solutions.

The cursor-based solution is straightforward. In fact, it's similar to calculating custom aggregates except for a simple difference: the code calculating custom aggregates set aside in a table variable only the final aggregate for each group, while the code calculating running aggregations sets aside the accumulated aggregate value for each row. Listing 3-3 shows the code that calculates running total quantities for each employee and month and yields the output shown in Table 3-2 (abbreviated).

Listing 3-3 Cursor code for running aggregations

```
DECLARE @Result
  TABLE(empid INT, ordmonth DATETIME, qty INT, runqty INT);
DECLARE
  @empid AS INT,@prvempid AS INT, @ordmonth DATETIME,
  @qty AS INT, @runqty AS INT;

DECLARE C CURSOR FAST_FORWARD FOR
  SELECT empid, ordmonth, qty
  FROM dbo.EmpOrders
  ORDER BY empid, ordmonth;

OPEN C
```

```
FETCH NEXT FROM C INTO @empid, @ordmonth, @qty;
SELECT @prvempid = @empid, @runqty = 0;

WHILE @@fetch_status = 0
BEGIN
  IF @empid <> @prvempid
    SELECT @prvempid = @empid, @runqty = 0;

  SET @runqty = @runqty + @qty;

  INSERT INTO @Result VALUES(@empid, @ordmonth, @qty, @runqty);

  FETCH NEXT FROM C INTO @empid, @ordmonth, @qty;
END

CLOSE C;

DEALLOCATE C;

SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth,
  qty, runqty
FROM @Result
ORDER BY empid, ordmonth;
```

Table 3-2 Running Aggregations (Abbreviated)

empid	*Ordmonth*	*qty*	*runqty*
1	1996-07	121	121
1	1996-08	247	368
1	1996-09	255	623
1	1996-10	143	766
1	1996-11	318	1084
1	1996-12	536	1620
1	1997-01	304	1924
1	1997-02	168	2092
1	1997-03	275	2367
1	1997-04	20	2387
…	…	…	…
2	1996-07	50	50
2	1996-08	94	144
2	1996-09	137	281
2	1996-10	248	529
2	1996-11	237	766
2	1996-12	319	1085
2	1997-01	230	1315
2	1997-02	36	1351

Table 3-2 Running Aggregations (Abbreviated)

empid	*Ordmonth*	*qty*	*runqty*
2	1997-03	151	1502
2	1997-04	468	1970
...	...	...	...

The cursor solution scans the data only once, meaning that it has linear performance degradation with respect to the number of rows in the table. The set-based solution suffers from an n^2 performance issue, in which n refers to the number of rows per group. If you have g groups with n number of rows per group, you scan $g \times (n + n^2)/2$ rows. This formula assumes that you have an index on *(groupid, val)*. Without an index, you simply have n^2 rows scanned, where n is the number of rows in the table. If the group size is small enough (for example, a dozen rows), the set-based solution that uses an index would typically be faster than the cursor solution. The cursor's overhead is still higher than the set-based solution's extra work of scanning more data. However, the set-based solution scans substantially more data (unless there are only a few rows per group), resulting in a slower solution where performance degrades in an n^2 manner with respect to the group size.

To gain a sense of these performance differences, look at Figure 3-1, which has the result of a benchmark.

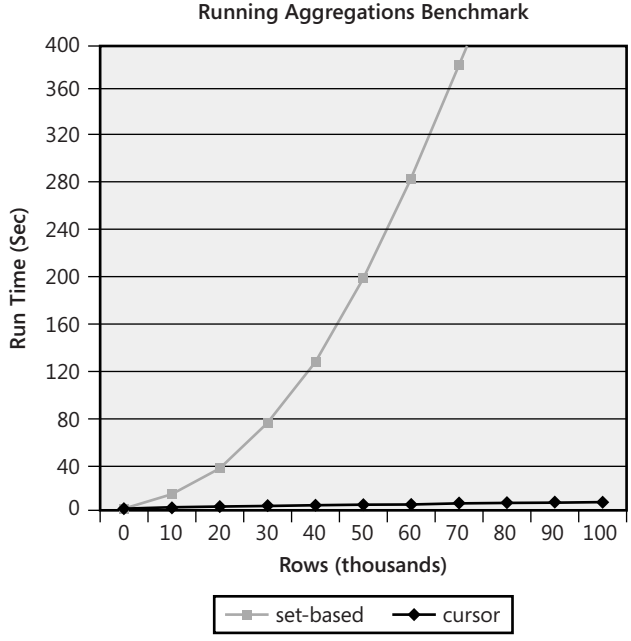

Figure 3-1 Benchmark for running calculations

You can see the run time of the solutions with respect to the number of rows in the table, assuming a single group—that is, by calculating running aggregations for the whole table. The horizontal axis has the number of rows in the table divided by 1000, ranging from 0 through

100,000 rows. The y axis ranges from 0 through 400 seconds. You can see a linear graph for the cursor solution, and a nice n^2 parabola for the set-based one. You can also notice clearly that beyond a very small number of rows the cursor solution performs dramatically faster.

This is one of the problems that ANSI already provided an answer for in the form of query constructs; however, SQL Server has not yet implemented it. According to ANSI, you would write the following solution:

```
SELECT empid, CONVERT(VARCHAR(7), ordmonth, 121) AS ordmonth, qty,
  SUM(qty) OVER(PARTITION BY empid ORDER BY ordermonth) AS runqty
FROM dbo.EmpOrders;
```

As mentioned earlier, SQL Server 2005 already introduced the infrastructure to support the OVER clause. It currently implements it with both the PARTITION BY and ORDER BY clauses for ranking functions, but only with the PARTITION BY clause for aggregate functions. Hopefully, future versions of SQL Server will enhance the support for the OVER clause. Queries such as the one just shown have the potential to run substantially faster than the cursor solution; the infrastructure added to the product relies on a single scan of the data to perform such calculations.

Maximum Concurrent Sessions

The Maximum Concurrent Sessions problem is yet another example of calculations based on ordered data. You record data for user sessions against different applications in a table called Sessions. Run the code in Listing 3-4 to create and populate the Sessions table.

Listing 3-4 Creating and populating the Sessions table

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Sessions') IS NOT NULL
  DROP TABLE dbo.Sessions;
GO

CREATE TABLE dbo.Sessions
(
  keycol    INT         NOT NULL IDENTITY PRIMARY KEY,
  app       VARCHAR(10) NOT NULL,
  usr       VARCHAR(10) NOT NULL,
  host      VARCHAR(10) NOT NULL,
  starttime DATETIME    NOT NULL,
  endtime   DATETIME    NOT NULL,
  CHECK(endtime > starttime)
);

INSERT INTO dbo.Sessions
  VALUES('app1', 'user1', 'host1', '20030212 08:30', '20030212 10:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user2', 'host1', '20030212 08:30', '20030212 08:45');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user3', 'host2', '20030212 09:00', '20030212 09:30');
```

```
INSERT INTO dbo.Sessions
  VALUES('app1', 'user4', 'host2', '20030212 09:15', '20030212 10:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user5', 'host3', '20030212 09:15', '20030212 09:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user6', 'host3', '20030212 10:30', '20030212 14:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user7', 'host4', '20030212 10:45', '20030212 11:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user8', 'host4', '20030212 11:00', '20030212 12:30');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user8', 'host1', '20030212 08:30', '20030212 08:45');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user7', 'host1', '20030212 09:00', '20030212 09:30');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user6', 'host2', '20030212 11:45', '20030212 12:00');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user5', 'host2', '20030212 12:30', '20030212 14:00');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user4', 'host3', '20030212 12:45', '20030212 13:30');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user3', 'host3', '20030212 13:00', '20030212 14:00');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user2', 'host4', '20030212 14:00', '20030212 16:30');
INSERT INTO dbo.Sessions
  VALUES('app2', 'user1', 'host4', '20030212 15:30', '20030212 17:00');

CREATE INDEX idx_app_st_et ON dbo.Sessions(app, starttime, endtime);
```

The request is to calculate, for each application, the maximum number of sessions that were open at the same point in time. Such types of calculations are required to determine the cost of a type of service license that charges by the maximum number of concurrent sessions.

Try to develop a set-based solution that works; then try to optimize it; and then try to estimate its performance potential. Later I'll discuss a cursor-based solution and show a benchmark that compares the set-based solution with the cursor-based solution.

One way to solve the problem is to generate an auxiliary table with all possible points in time during the covered period, use a subquery to count the number of active sessions during each such point in time, create a derived table/CTE from the result table, and finally group the rows from the derived table by application, requesting the maximum count of concurrent sessions for each application. Such a solution is extremely inefficient. Assuming you create the optimal index for it—one on (*app, starttime, endtime*)—the total number of rows you end up scanning just in the leaf level of the index is huge. It's equal to the number of rows in the auxiliary table multiplied by the average number of active sessions at any point in time. To give you a sense of the enormity of the task, if you need to perform the calculations for a month's worth of activity, the number of rows in the auxiliary table will be: 31 (days) × 24 (hours) × 60 (minutes) × 60 (seconds) × 300 (units within a second). Now multiply the result of this calculation by the average number of active sessions at any given point in time (say 20 as an example), and you get 16,070,400,000.

Of course there's room for optimization. There are periods in which the number of concurrent sessions doesn't change, so why calculate the counts for those? The count changes only when a new session starts (increased by 1) or an existing session ends (decreased by 1). Furthermore, because a start of a session increases the count and an end of a session decreases it, a start event of one of the sessions is bound to be the point at which you will find the maximum you're looking for. Finally, if two sessions start at the same time, there's no reason to calculate the counts for both. So you can apply a DISTINCT clause in the query that returns the start times for each application, although with an accuracy level of 3 ⅓ milliseconds (ms), the number of duplicates would be very small—unless you're dealing with very large volumes of data.

In short, you can simply use as your auxiliary table a derived table or CTE that returns all distinct start times of sessions per application. From there, all you need to do is follow logic similar to that mentioned earlier. Here's the optimized set-based solution, yielding the output shown in Table 3-3:

```
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app,
        (SELECT COUNT(*)
         FROM dbo.Sessions AS S2
         WHERE S1.app = S2.app
           AND S1.ts >= S2.starttime
           AND S1.ts < S2.endtime) AS concurrent
      FROM (SELECT DISTINCT app, starttime AS ts
            FROM dbo.Sessions) AS S1) AS C
GROUP BY app;
```

Table 3-3 Maximum Concurrent Sessions Set-Based Solution

app	*mx*
app1	4
app2	3

Notice that instead of using a BETWEEN predicate to determine whether a session was active at a certain point in time (*ts*), I used *ts >= starttime AND ts < endtime*. If a session ends at the *ts* point in time, I don't want to consider it as active.

The execution plan for this query is shown in Figure 3-2.

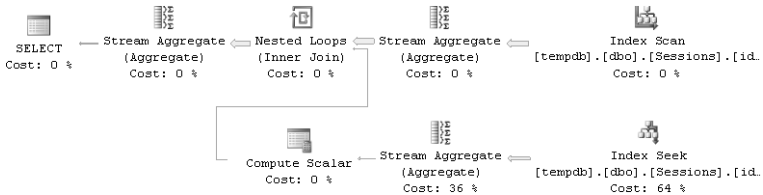

Figure 3-2 Execution plan for Maximum Concurrent Sessions, set-based solution

First, the index I created on (*app, starttime, endtime*) is scanned and duplicates are removed (by the stream aggregate operator). Unless the table is huge, you can assume that the number

of rows returned will be very close to the number of rows in the table. For each *app, starttime* (call it *ts*) returned after removing duplicates, a Nested Loops operator initiates activity that calculates the count of active sessions (by a seek within the index, followed by a partial scan to count active sessions). The number of pages read in each iteration of the Nested Loops operator is the number of levels in the index plus the number of pages consumed by the number of active sessions. To make my point, I'll focus on the number of rows scanned at the leaf level because this number varies based on active sessions. Of course, to do adequate performance estimations, you should take page counts (logical reads) as well as many other factors into consideration. If you have *n* rows in the table, assuming that most of them have unique *app, starttime* values and there are *o* overlapping sessions at any given point in time, you're looking at the following: $n \times o$ rows scanned in total at the leaf level, beyond the pages scanned by the seek operations that got you to the leaf.

You now need to figure out how this solution scales when the table grows larger. Typically, such reports are required periodically—for example, once a month, for the most recent month. With the recommended index in place, the performance shouldn't change as long as the traffic doesn't increase for a month's worth of activity—that is, if it's related to $n \times o$ (where *n* is the number of rows for the recent month). But suppose that you anticipate traffic increase by a factor of *f*? If traffic increases by a factor of f, both total rows and number of active sessions at a given time grow by that factor; so in total, the number of rows scanned at the leaf level becomes $(n \times f)(o \times f) = n \times o \times f^2$. You see, as the traffic grows, performance doesn't degrade linearly; rather, it degrades much more drastically.

Next let's talk about a cursor-based solution. The power of a cursor-based solution is that it can scan data in order. Relying on the fact that each session represents two events—one that increases the count of active sessions, and one that decreases the count—I'll declare a cursor for the following query:

```
SELECT app, starttime AS ts, 1 AS event_type FROM dbo.Sessions
UNION ALL
SELECT app, endtime, -1 FROM dbo.Sessions
ORDER BY app, ts, event_type;
```

This query returns the following for each session start or end event: the application (*app*), the timestamp (*ts*); an event type (*event_type*) of +1 for a session start event or −1 for a session end event. The events are sorted by *app*, *ts*, and *event_type*. The reason for sorting by *app, ts* is obvious. The reason for adding *event_type* to the sort is to guarantee that if a session ends at the same time another session starts, you will take the end event into consideration first (because sessions are considered to have ended at their end time). Other than that, the cursor code is straightforward—simply scan the data in order and keep adding up the +1s and −1s for each application. With every new row scanned, check whether the cumulative value to that point is greater than the current maximum for that application, which you store in a variable. If it is, store it as the new maximum. When done with an application, insert a row containing the application ID and maximum into a table variable. That's about it. You can find the complete cursor solution in Listing 3-5.

Listing 3-5 Cursor code for Maximum Concurrent Sessions, cursor-based solution

```
DECLARE
  @app AS VARCHAR(10), @prevapp AS VARCHAR (10), @ts AS datetime,
  @event_type AS INT, @concurrent AS INT, @mx AS INT;

DECLARE @Result TABLE(app VARCHAR(10), mx INT);

DECLARE C CURSOR FAST_FORWARD FOR
  SELECT app, starttime AS ts, 1 AS event_type FROM dbo.Sessions
  UNION ALL
  SELECT app, endtime, -1 FROM dbo.Sessions
  ORDER BY app, ts, event_type;

OPEN C;

FETCH NEXT FROM C INTO @app, @ts, @event_type;
SELECT @prevapp = @app, @concurrent = 0, @mx = 0;

WHILE @@fetch_status = 0
BEGIN
  IF @app <> @prevapp
  BEGIN
    INSERT INTO @Result VALUES(@prevapp, @mx);
    SELECT @prevapp = @app, @concurrent = 0, @mx = 0;
  END

  SET @concurrent = @concurrent + @event_type;
  IF @concurrent > @mx SET @mx = @concurrent;

  FETCH NEXT FROM C INTO @app, @ts, @event_type;
END

IF @prevapp IS NOT NULL
  INSERT INTO @Result VALUES(@prevapp, @mx);

CLOSE C

DEALLOCATE C

SELECT * FROM @Result;
```

The cursor solution scans the leaf of the index only twice. You can represent its cost as $n \times 2 \times v$, where v is the cursor overhead involved with each single row manipulation. Also, if the traffic grows by a factor of f, the performance degrades linearly to $n \times 2 \times v \times f$. You realize that unless you're dealing with a very small input set, the cursor solution has the potential to perform much faster, and as proof, you can use the code in Listing 3-6 to conduct a benchmark test. Change the value of the *@numrows* variable to determine the number of rows in the table. I ran this code with numbers varying from 10,000 through 100,000 in steps of 10,000. Figure 3-3 shows a graphical depiction of the benchmark test I ran.

Listing 3-6 Benchmark code for Maximum Concurrent Sessions problem

```
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.Sessions') IS NOT NULL
  DROP TABLE dbo.Sessions
GO

DECLARE @numrows AS INT;
SET @numrows = 10000;
-- Test with 10K - 100K

SELECT
  IDENTITY(int, 1, 1) AS keycol,
  D.*,
  DATEADD(
    second,
    1 + ABS(CHECKSUM(NEWID())) % (20*60),
    starttime) AS endtime
INTO dbo.Sessions
FROM
(
  SELECT
    'app' + CAST(1 + ABS(CHECKSUM(NEWID())) % 10 AS VARCHAR(10)) AS app,
    'user1' AS usr,
    'host1' AS host,
    DATEADD(
      second,
      1 + ABS(CHECKSUM(NEWID())) % (30*24*60*60),
      '20040101') AS starttime
  FROM dbo.Nums
  WHERE n <= @numrows
) AS D;

ALTER TABLE dbo.Sessions ADD PRIMARY KEY(keycol);
CREATE INDEX idx_app_st_et ON dbo.Sessions(app, starttime, endtime);

DBCC FREEPROCCACHE WITH NO_INFOMSGS;
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

DECLARE @dt1 AS DATETIME, @dt2 AS DATETIME,
  @dt3 AS DATETIME, @dt4 AS DATETIME;
SET @dt1 = GETDATE();

-- Set-Based Solution
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app,
        (SELECT COUNT(*)
         FROM dbo.Sessions AS S2
         WHERE S1.app = S2.app
           AND S1.ts >= S2.starttime
           AND S1.ts < S2.endtime) AS concurrent
      FROM (SELECT DISTINCT app, starttime AS ts
            FROM dbo.Sessions) AS S1) AS C
```

```
GROUP BY app;

SET @dt2 = GETDATE();

DBCC FREEPROCCACHE WITH NO_INFOMSGS;
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

SET @dt3 = GETDATE();

-- Cursor-Based Solution
DECLARE
  @app AS VARCHAR(10), @prevapp AS VARCHAR (10), @ts AS datetime,
  @event_type AS INT, @concurrent AS INT, @mx AS INT;

DECLARE @Result TABLE(app VARCHAR(10), mx INT);

DECLARE C CURSOR FAST_FORWARD FOR
  SELECT app, starttime AS ts, 1 AS event_type FROM dbo.Sessions
  UNION ALL
  SELECT app, endtime, -1 FROM dbo.Sessions
  ORDER BY app, ts, event_type;

OPEN C;

FETCH NEXT FROM C INTO @app, @ts, @event_type;
SELECT @prevapp = @app, @concurrent = 0, @mx = 0;

WHILE @@fetch_status = 0
BEGIN
  IF @app <> @prevapp
  BEGIN
    INSERT INTO @Result VALUES(@prevapp, @mx);
    SELECT @prevapp = @app, @concurrent = 0, @mx = 0;
  END

  SET @concurrent = @concurrent + @event_type;
  IF @concurrent > @mx SET @mx = @concurrent;

  FETCH NEXT FROM C INTO @app, @ts, @event_type;
END

IF @prevapp IS NOT NULL
  INSERT INTO @Result VALUES(@prevapp, @mx);

CLOSE C

DEALLOCATE C

SELECT * FROM @Result;

SET @dt4 = GETDATE();

PRINT CAST(@numrows AS VARCHAR(10)) + ' rows, set-based: '
  + CAST(DATEDIFF(ms, @dt1, @dt2) / 1000. AS VARCHAR(30))
  + ', cursor: '
  + CAST(DATEDIFF(ms, @dt3, @dt4) / 1000. AS VARCHAR(30))
  + ' (sec)';
```

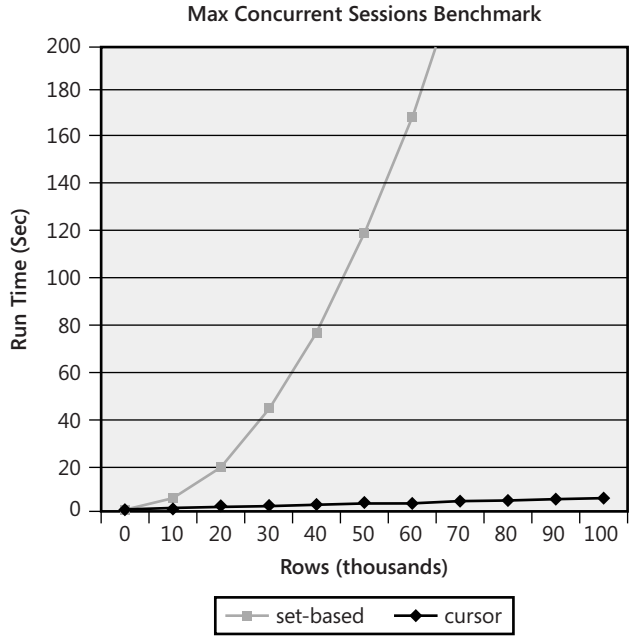

Figure 3-3 Benchmark for Maximum Concurrent Sessions solutions

Again, you can see a nicely shaped parabola in the set-based solution's graph, and now you know how to explain it: remember—if traffic increases by a factor of *f*, the number of leaf-level rows inspected by the set-based query grows by a factor of f^2.

Tip It might seem that all the cases in which I show cursor code that performs better than set-based code have to do with problems where cursor code has a complexity of *O(n)* and set-based code has a complexity of $O(n^2)$, where *n* is the number of rows in the table. These are just convenient problems to demonstrate performance differences. However, you might face problems for which the solutions have different complexities. The important point is to be able to estimate complexity and performance. If you want to learn more about algorithmic complexity, visit the Web site of the National Institute for Standards and Technologies. Go to *http://www.nist.gov/dads/*, and search for complexity, or access the definition directly at *http://www.nist.gov/dads/HTML/complexity.html*.

Interestingly, this is yet another type of problem where a more complete implementation of the OVER clause would have allowed for a set-based solution to perform substantially faster than the cursor one. Here's what the set-based solution would have looked like if SQL Server supported ORDER BY in the OVER clause for aggregations:

```
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app, SUM(event_type)
        OVER(PARTITION BY app ORDER BY ts, event_type) AS concurrent
      FROM (SELECT app, starttime AS ts, 1 AS event_type FROM dbo.Sessions
            UNION ALL
            SELECT app, endtime, -1 FROM dbo.Sessions) AS D1) AS D2
GROUP BY app;
```

Before I proceed to the next class of problems, I'd like to stress the importance of using good sample data in your benchmarks. Too often I have seen programmers simply duplicate data from a small table many times to generate larger sets of sample data. With our set-based solution, remember the derived table query that generates the timestamps:

```
SELECT DISTINCT app, starttime AS ts
FROM dbo.Sessions
```

If you simply duplicate the small sample data that I provided in Listing 3-4 (16 rows) many times, you will not increase the number of DISTINCT timestamps accordingly. So the subquery that counts rows will end up being invoked only 16 times regardless of how many times you duplicated the set. The results that you will get when measuring performance won't give you a true indication of cost for production environments where, obviously, you have almost no duplicates in the data.

The solution to the problem can be even more elusive if you don't have any DISTINCT applied to remove duplicates. To demonstrate the problem, first rerun the code in Listing 3-4 to repopulate the Sessions table with 16 rows.

Next, run the following query, which is similar to the solution I showed earlier, but run it without removing duplicates first. Then examine the execution plan shown in Figure 3-4:

```
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app,
        (SELECT COUNT(*)
         FROM dbo.Sessions AS S2
         WHERE S1.app = S2.app
           AND S1.starttime >= S2.starttime
           AND S1.starttime < S2.endtime) AS concurrent
      FROM dbo.Sessions AS S1) AS C
GROUP BY app;
```

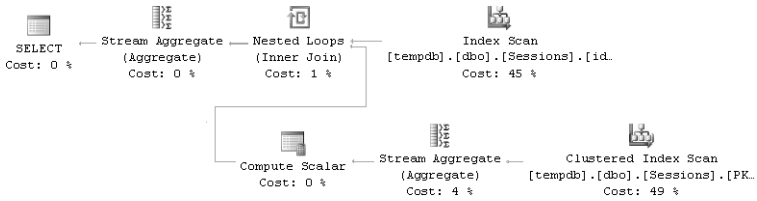

Figure 3-4 Execution plan for revised Maximum Concurrent Sessions solution, small data set

Here the problem is not yet apparent because there are no duplicates. The plan is, in fact, almost identical to the one generated for the solution that does remove duplicates. The only difference is that here there's no stream aggregate operator that removes duplicates, naturally.

Next, populate the table with 10,000 duplicates of each row:

```
INSERT INTO dbo.Sessions
  SELECT app, usr, host, starttime, endtime
  FROM dbo.Sessions, dbo.Nums
  WHERE n <= 10000;
```

Rerun the solution query, and examine the execution plan shown in Figure 3-5.

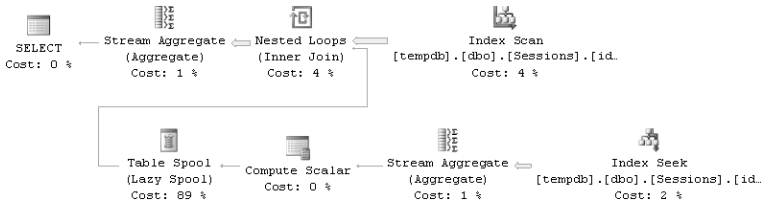

Figure 3-5 Execution plan for revised Maximum Concurrent Sessions solution, large data set with high density

If you have a keen eye, you will find an interesting difference between this plan and the previous one, even though the query remained the same and only the data density changed. This plan spools, instead of recalculating, row counts that were already calculated for a given *app, ts*. Before counting rows, the plan first looks in the spool to check whether the count has already been calculated. If the count has been calculated, the plan will grab the count from the spool instead of scanning rows to count. The Index Seek and Stream Aggregate operations took place here only 16 times—once for each unique *app, ts* value, and not once for each row in the table as might happen in production. Again, you see how a bad choice of sample data can yield a result that is not representative of your production environment. Using this sample data and being oblivious to the discrepancy might lead you to believe that this set-based solution scales linearly. But of course, if you use more realistic sample data, such as the data I used in my benchmark, you won't fall into that trap. I used random calculations for the start times within the month and added a random value of up to 20 minutes for the end time, assuming that this represents the average session duration in my production environment.

Matching Problems

The algorithms for the solutions that I have discussed so far, both set-based and cursor-based, had simple to moderate complexity levels. This section covers a class of problems that are algorithmically much more complex, known as *matching problems*. In a matching problem, you have a specific set of items of different values and volumes and one container of a given size, and you must find the subset of items with the greatest possible value that will fit into the container. I have yet to find reasonable set-based solutions that are nearly as good as cursor-based solutions, both in terms of performance and simplicity. I won't even bother to provide the set-based solutions I devised because they're very complex and slow. Instead, I'll focus on cursor-based solutions.

I'll introduce a couple of simple variations of the problem. You're given the tables Events and Rooms, which you create and populate by running the code in Listing 3-7.

Listing 3-7 Code that creates and populates the Events and Rooms tables

```
USE tempdb;
GO
IF OBJECT_ID('dbo.Events') IS NOT NULL
  DROP TABLE dbo.Events;
GO
IF OBJECT_ID('dbo.Rooms') IS NOT NULL
  DROP TABLE dbo.Rooms;
GO

CREATE TABLE dbo.Rooms
(
  roomid VARCHAR(10) NOT NULL PRIMARY KEY,
  seats INT NOT NULL
);

INSERT INTO dbo.Rooms(roomid, seats) VALUES('C001', 2000);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('B101', 1500);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('B102', 100);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('R103', 40);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('R104', 40);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('B201', 1000);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('R202', 100);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('R203', 50);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('B301', 600);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('R302', 55);
INSERT INTO dbo.Rooms(roomid, seats) VALUES('R303', 55);

CREATE TABLE dbo.Events
(
  eventid INT NOT NULL PRIMARY KEY,
  eventdesc VARCHAR(25) NOT NULL,
  attendees INT NOT NULL
);

INSERT INTO dbo.Events(eventid, eventdesc, attendees)
  VALUES(1, 'Adv T-SQL Seminar', 203);
INSERT INTO dbo.Events(eventid, eventdesc, attendees)
  VALUES(2, 'Logic Seminar',     48);
INSERT INTO dbo.Events(eventid, eventdesc, attendees)
  VALUES(3, 'DBA Seminar',       212);
INSERT INTO dbo.Events(eventid, eventdesc, attendees)
  VALUES(4, 'XML Seminar',       98);
INSERT INTO dbo.Events(eventid, eventdesc, attendees)
  VALUES(5, 'Security Seminar',  892);
INSERT INTO dbo.Events(eventid, eventdesc, attendees)
  VALUES(6, 'Modeling Seminar',  48);
GO

CREATE INDEX idx_att_eid_edesc
  ON dbo.Events(attendees, eventid, eventdesc);
CREATE INDEX idx_seats_rid
  ON dbo.Rooms(seats, roomid);
```

The Events table holds information for seminars that you're supposed to run on a given date. Typically, you will need to keep track of events on many dates, but our task here will be one that we would have to perform separately for each day of scheduled events. Assume that this data represents one day's worth of events; for simplicity's sake, I didn't include a date column because all its values would be the same. The Rooms table holds room capacity information. To start with a simple task, assume that you have reserved a conference center with the guarantee that there will be enough rooms available to host all your seminars. You now need to match events to rooms with as few empty seats as possible, because the cost of renting a room is determined by the room's seating capacity, not by the number of seminar attendees.

A naïve algorithm that you can apply is somewhat similar to a merge join algorithm that the optimizer uses to process joins. Figure 3-6 has a graphical depiction of it, which you might find handy when following the verbal description of the algorithm. Listing 3-8 has the code implementing the algorithm.

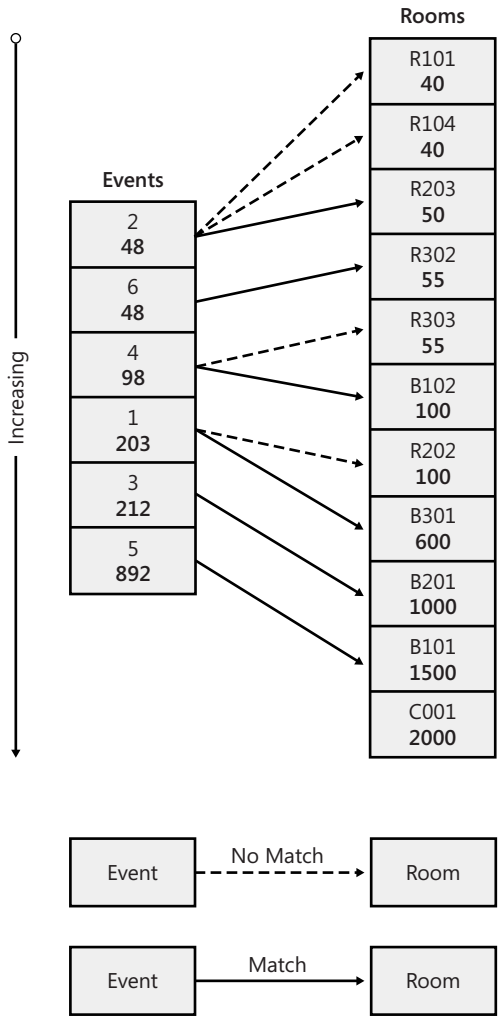

Figure 3-6 Matching algorithm for guaranteed solution scenario

Listing 3-8 Cursor code for matching problem (guaranteed solution)

```
DECLARE
  @roomid AS VARCHAR(10), @seats AS INT,
  @eventid AS INT, @attendees AS INT;

DECLARE @Result TABLE(roomid  VARCHAR(10), eventid INT);

DECLARE CRooms CURSOR FAST_FORWARD FOR
  SELECT roomid, seats FROM dbo.Rooms
  ORDER BY seats, roomid;
DECLARE CEvents CURSOR FAST_FORWARD FOR
  SELECT eventid, attendees FROM dbo.Events
  ORDER BY attendees, eventid;

OPEN CRooms;
OPEN CEvents;

FETCH NEXT FROM CEvents INTO @eventid, @attendees;
WHILE @@FETCH_STATUS = 0
BEGIN
  FETCH NEXT FROM CRooms INTO @roomid, @seats;

  WHILE @@FETCH_STATUS = 0 AND @seats < @attendees
    FETCH NEXT FROM CRooms INTO @roomid, @seats;

  IF @@FETCH_STATUS = 0
    INSERT INTO @Result(roomid, eventid) VALUES(@roomid, @eventid);
  ELSE
  BEGIN
    RAISERROR('Not enough rooms for events.', 16, 1);
    BREAK;
  END

  FETCH NEXT FROM CEvents INTO @eventid, @attendees;
END

CLOSE CRooms;
CLOSE CEvents;

DEALLOCATE CRooms;
DEALLOCATE CEvents;

SELECT roomid, eventid FROM @Result;
```

Here's a description of the algorithm as it's implemented with cursors:

- Declare two cursors, one on the list of rooms (*CRooms*) sorted by increasing capacity (number of seats), and one on the list of events (*CEvents*) sorted by increasing number of attendees.
- Fetch the first (smallest) event from the *CEvents* cursor.

- While the fetch returned an actual event that needs a room:
  - Fetch the smallest unrented room from *CRooms*. If there was no available room, or if the room you fetched is too small for the event, fetch the next smallest room from *CRooms*, and continue fetching as long as you keep fetching actual rooms and they are too small for the event. You will either find a big enough room, or you will run out of rooms without finding one.
  - If you did not run out of rooms, and the last fetch yielded a room and the number of seats in that room is smaller than the number of attendees in the current event:
    - If you found a big enough room, schedule the current event in that room. If you did not, then you must have run out of rooms, so generate an error saying that there are not enough rooms to host all the events, and break out of the loop.
    - Fetch another event.
- Return the room/event pairs you stored aside.

Notice that you scan both rooms and events in order, never backing up; you merge matching pairs until you either run out of events to find rooms for or you run out of rooms to accommodate events. In the latter case—you run out of rooms, generating an error, because the algorithm used was guaranteed to find a solution if one existed.

Next, let's complicate the problem by assuming that even if there aren't enough rooms for all events, you still want to schedule something. This will be the case if you remove rooms with a number of seats greater than 600:

```
DELETE FROM dbo.Rooms WHERE seats > 600;
```

Assume you need to come up with a *greedy* algorithm that finds seats for the highest possible number of attendees (to increase revenue) and for that number of attendees, involves the lowest cost. The algorithm I used for this case is graphically illustrated in Figure 3-7 and implemented with cursors in Listing 3-9.

Listing 3-9 Cursor code for matching problem (nonguaranteed solution)

```
DECLARE
  @roomid AS VARCHAR(10), @seats AS INT,
  @eventid AS INT, @attendees AS INT;

DECLARE @Events TABLE(eventid INT, attendees INT);
DECLARE @Result TABLE(roomid  VARCHAR(10), eventid INT);

-- Step 1: Descending
DECLARE CRoomsDesc CURSOR FAST_FORWARD FOR
  SELECT roomid, seats FROM dbo.Rooms
  ORDER BY seats DESC, roomid DESC;
DECLARE CEventsDesc CURSOR FAST_FORWARD FOR
  SELECT eventid, attendees FROM dbo.Events
  ORDER BY attendees DESC, eventid DESC;
```

```
OPEN CRoomsDesc;
OPEN CEventsDesc;

FETCH NEXT FROM CRoomsDesc INTO @roomid, @seats;
WHILE @@FETCH_STATUS = 0
BEGIN
  FETCH NEXT FROM CEventsDesc INTO @eventid, @attendees;

  WHILE @@FETCH_STATUS = 0 AND @seats < @attendees
    FETCH NEXT FROM CEventsDesc INTO @eventid, @attendees;

  IF @@FETCH_STATUS = 0
    INSERT INTO @Events(eventid, attendees)
      VALUES(@eventid, @attendees);
  ELSE
    BREAK;

  FETCH NEXT FROM CRoomsDesc INTO @roomid, @seats;
END

CLOSE CRoomsDesc;
CLOSE CEventsDesc;

DEALLOCATE CRoomsDesc;
DEALLOCATE CEventsDesc;

-- Step 2: Ascending
DECLARE CRooms CURSOR FAST_FORWARD FOR
  SELECT roomid, seats FROM Rooms
  ORDER BY seats, roomid;
DECLARE CEvents CURSOR FAST_FORWARD FOR
  SELECT eventid, attendees FROM @Events
  ORDER BY attendees, eventid;

OPEN CRooms;
OPEN CEvents;

FETCH NEXT FROM CEvents INTO @eventid, @attendees;
WHILE @@FETCH_STATUS = 0
BEGIN
  FETCH NEXT FROM CRooms INTO @roomid, @seats;

  WHILE @@FETCH_STATUS = 0 AND @seats < @attendees
    FETCH NEXT FROM CRooms INTO @roomid, @seats;

  IF @@FETCH_STATUS = 0
    INSERT INTO @Result(roomid, eventid) VALUES(@roomid, @eventid);
  ELSE
  BEGIN
    RAISERROR('Not enough rooms for events.', 16, 1);
    BREAK;
  END

  FETCH NEXT FROM CEvents INTO @eventid, @attendees;
END
```

```
CLOSE CRooms;
CLOSE CEvents;

DEALLOCATE CRooms;
DEALLOCATE CEvents;

SELECT roomid, eventid FROM @Result;
```

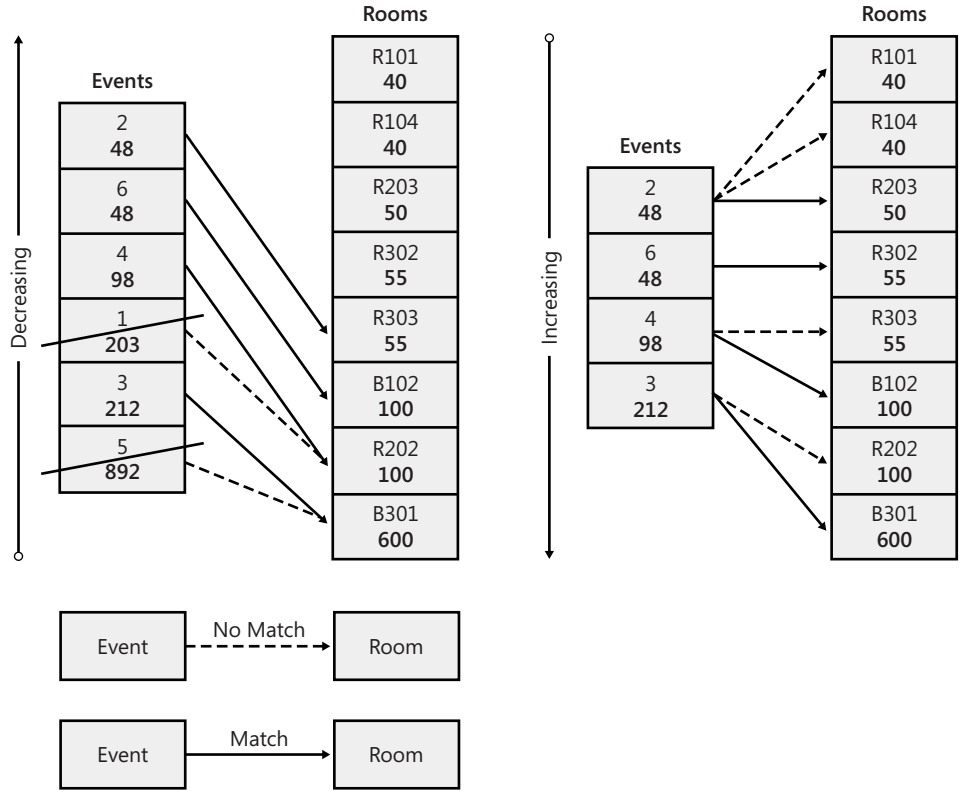

Figure 3-7 Greedy matching algorithm for nonguaranteed solution scenario

The algorithm has two phases:

1. Use logic similar to the previous algorithm to match events to rooms, but scan both in descending order to assure the largest events can find rooms. Store the *eventid*s that found a room in a table variable (*@Events*). At this point, you have the list of events you can fit that produce the highest revenue, but you also have the least efficient room utilization, meaning the highest possible costs. However, the purpose of the first step was merely to figure out the most profitable events that you can accommodate.
2. The next step is identical to the algorithm in the previous problem with one small revision: declare the *CEvents* cursor against the *@Events* table variable and not against the real Events table. By doing this, you end up with the most efficient room utilization for this set of events.

I'd like to thank my good friend, SQL Server MVP Fernando G. Guerrero, who is the CEO of Solid Quality Learning. Fernando suggested ways to improve and optimize the algorithms for this class of problems.

If you're up for challenges, try to look for ways to solve these problems with set-based solutions. Also, try to think of solutions when adding another layer of complexity. Suppose each event has a revenue value stored with it that does not necessarily correspond to the number of attendees. Each room has a cost stored with it that does not necessarily correspond to its capacity. Again, you have no guarantee that there will be enough rooms to host all events. The challenge is to find the most profitable solution.

Conclusion

Throughout the book, I try to stress the advantages set-based solutions have over cursor-based ones. I show many examples of tuned set-based solutions that outperform the cursor alternatives. In this chapter, I explained why that's the case for most types of problems. Nevertheless, I tried giving you the tools to identify the classes of problems that are exceptions—where currently SQL Server 2005 doesn't provide a better solution than using cursors. Some of the problems would have better set-based answers if SQL Server implemented additional ANSI constructs, whereas others don't even have proper answers in the ANSI standard yet. The point is that there's a time and place for cursors if they are used wisely and if a set-based means of solving the problem cannot be found.

Chapter 7

Stored Procedures

In this chapter:

Stored procedures are executable server-side routines. They give you great power and performance benefits if used wisely. Unlike user-defined functions (UDFs), stored procedures are allowed to have side effects. That is, they are allowed to change data in tables, and even the schema of objects. Stored procedures can be used as a security layer. You can control access to objects by granting execution permissions on stored procedures and not to underlying objects. You can perform input validation in stored procedures, and you can use stored procedures to allow activities only if they make sense as a whole unit, as opposed to allowing users to perform activities directly against objects.

Stored procedures also give you the benefits of encapsulation; if you need to change the implementation of a stored procedure because you developed a more efficient way to achieve a task, you can issue an ALTER PROCEDURE statement. As long as the procedure's interface remains the same, the users and the applications are not affected. On the other hand, if you implement your business logic in the client application, the impact of a change can be very painful.

Stored procedures also provide many important performance benefits. By default, a stored procedure will reuse a previously cached execution plan, saving the CPU resources and the time it takes to parse, resolve, and optimize your code. Network traffic is minimized by shortening the code strings that the client submits to Microsoft SQL Server—the client submits only the stored procedure's name and its arguments, as opposed to the full code. Moreover, all the activity is performed at the server, avoiding multiple roundtrips between the client and the server. The stored procedure will pass only the final result to the client through the network.

This chapter explores stored procedures. It starts with brief coverage of the different types of stored procedures supported by SQL Server 2005 and then delves into details. The chapter covers the stored procedure's interface, resolution process, compilation, recompilations and execution plan reuse, the EXECUTE AS clause, and the new common language runtime (CLR) stored procedures. You will have a couple of chances in the chapter to practice what you've learned by developing stored procedures that serve common practical needs.

Types of Stored Procedures

SQL Server 2005 supports different types of stored procedures: user-defined, system, and extended. You can develop user-defined stored procedures with T-SQL or with the CLR. This section briefly covers the different types.

User-Defined Stored Procedures

A user-defined stored procedure is created in a user database and typically interacts with the database objects. When you invoke a user-defined stored procedure, you specify the EXEC (or EXECUTE) command and the stored procedure's schema-qualified name, and arguments:

```
EXEC dbo.usp_Proc1 <arguments>;
```

As an example, run the code in Listing 7-1 to create the usp_GetSortedShippers stored procedure in the Northwind database:

Listing 7-1 Creation Script for usp_GetSortedShippers

```
USE Northwind;
GO
IF OBJECT_ID('dbo.usp_GetSortedShippers') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers;
GO
-- Stored procedure usp_GetSortedShippers
-- Returns shippers sorted by requested sort column
CREATE PROC dbo.usp_GetSortedShippers
  @colname AS sysname = NULL
AS

DECLARE @msg AS NVARCHAR(500);

-- Input validation
IF @colname IS NULL
BEGIN
  SET @msg = N'A value must be supplied for parameter @colname.';
  RAISERROR(@msg, 16, 1);
  RETURN;
END
```

```
IF @colname NOT IN(N'ShipperID', N'CompanyName', N'Phone')
BEGIN
  SET @msg =
    N'Valid values for @colname are: '
    + N'N''ShipperID'', N''CompanyName'', N''Phone''.';
  RAISERROR(@msg, 16, 1);
  RETURN;
END

-- Return shippers sorted by requested sort column
IF @colname = N'ShipperID'
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY ShipperID;
ELSE IF @colname = N'CompanyName'
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY CompanyName;
ELSE IF @colname = N'Phone'
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY Phone;
GO
```

The stored procedure accepts a column name from the Shippers table in the Northwind database as input (*@colname*); after input validation, it returns the rows from the Shippers table sorted by the specified column name. Input validation here involves verifying that a column name was specified, and that the specified column name exists in the Shippers table. Later in the chapter, I will discuss the subject of parameterizing sort order in more detail; for now, I just wanted to provide a simple example of a user-defined stored procedure. Run the following code to invoke usp_GetSortedShippers specifying *N'CompanyName'* as input, generating the output shown in Table 7-1:

```
USE Northwind;
EXEC dbo.usp_GetSortedShippers @colname = N'CompanyName';
```

Table 7-1 Shippers Sorted by *CompanyName*

ShipperID	CompanyName	Phone
3	Federal Shipping	(503) 555-9931
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199

You can leave out the keyword EXEC if the stored procedure is the first statement of a batch, but I recommend using it all the time. You can also omit the stored procedure's schema name (*dbo* in our case), but when you neglect to specify it, SQL Server must resolve the schema. The resolution in SQL Server 2005 occurs in the following order (adapted from Books Online):

- The sys schema of the current database.

- The caller's default schema if executed in a batch or in dynamic SQL. Or, if the nonqualified procedure name appears inside the body of another procedure definition, the schema containing this other procedure is searched next.
- The dbo schema in the current database.

As an example, suppose that you connect to the Northwind database and your user's default schema in Northwind is called schema1. You invoke the following code in a batch:

```
EXEC usp_GetSortedShippers @colname = N'CompanyName';
```

The resolution takes place in the following order:

- Look for usp_GetSortedShippers in the sys schema of Northwind (sys.usp_GetSortedShippers). If found, execute it; if not, proceed to the next step (as in our case).
- If invoked in a batch (as in our case) or dynamic SQL, look for usp_GetSortedShippers in schema1 (schema1.usp_GetSortedShippers). Or, if invoked in another procedure (say, schema2.usp_AnotherProc), look for usp_GetSortedShippers in schema2 next. If found, execute it; if not, proceed to the next step (as in our case).
- Look for usp_GetSortedShippers in the dbo schema (dbo.usp_GetSortedShippers). If found (as in our case), execute it; if not, generate a resolution error.

Besides the potential for confusion and ambiguity when not specifying the schema, there's also an important performance reason to always specify it. When many connections are simultaneously running the same stored procedure, they may begin to block each other due to compile locks that they need to obtain when the schema name is not specified.

More Info For more information about this problem, please refer to Knowledge Base Article ID 263889, "Description of SQL blocking caused by compile locks," at *http://support.microsoft.com/?id=263889*.

As I mentioned earlier, stored procedures can be used as a security layer. You can control access to objects by granting execution permissions on stored procedures and not to underlying objects. For example, suppose that there's a database user called user1 in the Northwind database. You want to allow user1 to invoke the usp_GetSortedShippers procedure, but you want to deny user1 from accessing the Shippers table directly. You can achieve this by granting the user with EXECUTE permissions on the procedure, and denying SELECT (and possibly other) permissions on the table, as in:

```
DENY SELECT ON dbo.Shippers TO user1;
GRANT EXECUTE ON dbo.usp_GetSortedShippers TO user1;
```

SQL Server will allow user1 to execute the stored procedure. However, if user1 attempts to query the Shippers table directly:

```
SELECT ShipperID, CompanyName, Phone
FROM dbo.Shippers;
```

SQL Server will generate the following error:

```
Msg 229, Level 14, State 5, Line 1
SELECT permission denied on object 'Shippers', database 'Northwind', schema 'dbo'.
```

This security model gives you a high level of control over the activities that users will be allowed to perform.

I'd like to point out other aspects of stored procedure programming through the usp_GetSortedShippers sample procedure:

- Notice that I explicitly specified column names in the query and didn't use SELECT *. Using SELECT * is a bad practice. In the future, the table might undergo schema changes that cause your application to break. Also, if you really need only a subset of the table's columns and not all of them, the use of SELECT * prevents the optimizer from utilizing covering indexes defined on that subset of columns.
- The query is missing a filter. This is not a bad practice by itself; this is perfectly valid if you really need all rows from the table. But you might be surprised to learn that in performance-tuning projects at Solid Quality Learning, we still find production applications that need filtered data but filter it only at the client. Such an approach introduces extreme pressure on both SQL Server and the network. Filters allow the optimizer to consider using indexes, which minimizes the I/O cost. Also, by filtering at the server, you reduce network traffic. If you need filtered data, make sure you filter it at the server; use a WHERE clause (or ON, HAVING where relevant)!
- Notice the use of a semicolon (;) to suffix statements. Although not a requirement of T-SQL for all statements, the semicolon suffix is an ANSI requirement. In SQL Server 2000, a semicolon is not required at all but is optional. In SQL Server 2005, you are required to suffix some statements with a semicolon to avoid ambiguity of your code. For example, the WITH keyword is used for different purposes—to define a CTE, to specify a table hint, and others. SQL Server requires you to suffix the statement preceding the CTE's WITH clause to avoid ambiguity. Getting used to suffixing all statements with a semicolon is a good practice.

Now let's get back to the focus of this section—user-defined stored procedures.

As I mentioned earlier, to invoke a user-defined stored procedure, you specify EXEC, the schema-qualified name of the procedure, and the parameter values for the invocation if there are any. References in the stored procedure to system and user object names that are not fully qualified (that is, without the database prefix) are always resolved in the database in which

the procedure was created. If you want to invoke a user-defined procedure created in another database, you must database-qualify its name. For example, if you are connected to a database called db1 and want to invoke a stored procedure called usp_Proc1, which resides in db2, you would use the following code:

```
USE db1;
EXEC db2.dbo.usp_Proc1 <arguments>;
```

Invoking a procedure from another database wouldn't change the fact that object names that are not fully qualified would be resolved in the database in which the procedure was created (db2, in this case).

If you want to invoke a remote stored procedure residing in another instance of SQL Server, you would use the fully qualified stored procedure name, including the linked server name: server.database.schema.proc.

When done, run the following code for cleanup:

```
USE Northwind;
GO
IF OBJECT_ID('dbo.usp_GetSortedShippers') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers;
```

Special Stored Procedures

By "special stored procedure," I mean a stored procedure created with a name beginning with *sp_* in the master database. A stored procedure created in this way has special behavior.

Important Note that Microsoft strongly recommends against creating your own stored procedures with the *sp_* prefix. This prefix is used by SQL Server to designate system stored procedures. In this section, I will create stored procedures with the *sp_* prefix to demonstrate their special behavior.

As an example, the following code creates the special procedure sp_Proc1, which prints the database context and queries the INFORMATION_SCHEMA.TABLES view—first with dynamic SQL, then with a static query:

```
SET NOCOUNT ON;
USE master;
GO

IF OBJECT_ID('dbo.sp_Proc1') IS NOT NULL
  DROP PROC dbo.sp_Proc1;
GO

CREATE PROC dbo.sp_Proc1
AS
PRINT 'master.dbo.sp_Proc1 executing in ' + DB_NAME();
```

```
-- Dynamic query
EXEC('SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = ''BASE TABLE'';');

-- Static query
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE';
GO
```

One of the unique aspects of a special procedure is that you don't need to database-qualify its name when connected to another database. For example, you can be connected to Northwind and still be able to run it without database-qualifying its name:

```
USE Northwind;
EXEC dbo.sp_Proc1;
```

The PRINT command returns *'master.dbo.sp_Proc1 executing in Northwind'*. The database name in the printed message was obtained by the DB_NAME function. It seems that DB_NAME "thinks" that the database context is Northwind (the current database) and not master. Similarly, dynamic SQL also assumes the context of the current database; so the EXEC command (which invokes a query against INFORMATION_SCHEMA.TABLES) returns table names from the Northwind database. In contrast to the previous two statements, the static query against INFORMATION_SCHEMA.TABLES seems to "think" that it is running in master—it returns table names from the master database and not Northwind. Similarly, if you refer with static code to user objects (for example, a table called T1), SQL Server will look for them in master. If that's not confusing enough, in SQL Server 2000, static code referring to system tables (for example, sysobjects) was resolved in the current database. SQL Server 2005 preserves this behavior with the corresponding backward compatibility views (for example, sys.sysobjects)—but not with the new catalog views (for example, sys.objects).

Interestingly, the *sp_* prefix works magic also with other types of objects besides stored procedures.

Caution The behavior described in the following section is undocumented, and you should not rely on it in production environments.

For example, the following code creates a table with the *sp_* prefix in master:

```
USE master;
GO
IF OBJECT_ID('dbo.sp_Globals') IS NOT NULL
  DROP TABLE dbo.sp_Globals;
GO

CREATE TABLE dbo.sp_Globals
(
  var_name sysname     NOT NULL PRIMARY KEY,
  val      SQL_VARIANT NULL
);
```

And the following code switches between database contexts, and it always manages to find the table even though the table name is not database-qualified.

```
USE Northwind;
INSERT INTO dbo.sp_Globals(var_name, val)
  VALUES('var1', 10);
USE pubs;
INSERT INTO dbo.sp_Globals(var_name, val)
  VALUES('var2', CAST(1 AS BIT));
USE tempdb;
SELECT var_name, val FROM dbo.sp_Globals;
```

The last query produces the output shown in Table 7-2.

Table 7-2 Contents of sp_Globals Table

var_name	*Val*
var1	10
var2	1

For cleanup, run the following code:

```
USE master;
GO
IF OBJECT_ID('dbo.sp_Globals') IS NOT NULL
  DROP TABLE dbo.sp_Globals;
```

Do not drop sp_Proc1 yet because it is used in the following section.

System Stored Procedures

System stored procedures are procedures that were shipped by Microsoft. In SQL Server 2000, system stored procedures resided in the master database, had the *sp_* prefix, and were marked with the "*system*" (MS Shipped) flag. In SQL Server 2005, system stored procedures reside physically in an internal hidden Resource database, and they exist logically in every database.

A special procedure (*sp_* prefix, created in master) that is also marked as a system procedure gets additional unique behavior. When the installation scripts that are run by SQL Server's setup program create system procedures, they mark those procedures as system using the undocumented procedure *sp_MS_marksystemobject*.

Caution You should not use the *sp_MS_marksystemobject* stored procedure in production because you won't get any support if you run into trouble with them. Also, there's no guarantee that the behavior you get by marking your procedures as system will remain the same in future versions of SQL Server, or even future service packs. Here, I'm going to use it for demonstration purposes to show additional behaviors that system procedures have.

Run the following code to mark the special procedure sp_Proc1 also as a system procedure:

```
USE master;
EXEC sp_MS_marksystemobject 'dbo.sp_Proc1';
```

If you now run sp_Proc1 in databases other than master, you will observe that all code statements within the stored procedure assume the context of the current database:

```
USE Northwind;
EXEC dbo.sp_Proc1;
USE pubs;
EXEC dbo.sp_Proc1;
EXEC Northwind.dbo.sp_Proc1;
```

As a practice, avoid using the *sp_* prefix for user-defined stored procedures. Remember that if a local database has a stored procedure with the same name and schema as a special procedure in master, the user-defined procedure will be invoked. To demonstrate this, create a procedure called sp_Proc1 in Northwind as well:

```
USE Northwind;
GO
IF OBJECT_ID('dbo.sp_Proc1') IS NOT NULL
  DROP PROC dbo.sp_Proc1;
GO

CREATE PROC dbo.sp_Proc1
AS
PRINT 'Northwind.dbo.sp_Proc1 executing in ' + DB_NAME();
GO
```

If you run the following code, you will observe that when connected to Northwind, sp_Proc1 from Northwind was invoked:

```
USE Northwind;
EXEC dbo.sp_Proc1;
USE pubs;
EXEC dbo.sp_Proc1;
```

Drop the Northwind version because it would interfere with the following examples:

```
USE Northwind;
GO
IF OBJECT_ID('dbo.sp_Proc1') IS NOT NULL
  DROP PROC dbo.sp_Proc1;
```

Interestingly, system procedures have an additional unique behavior. They also resolve user objects in the current database, not just system objects. To demonstrate this, run the

following code to re-create the sp_Proc1 special procedure, which queries a user table called Orders, and to mark the procedure as system:

```
USE master;
GO
IF OBJECT_ID('dbo.sp_Proc1') IS NOT NULL
  DROP PROC dbo.sp_Proc1;
GO

CREATE PROC dbo.sp_Proc1
AS
PRINT 'master.dbo.sp_Proc1 executing in ' + DB_NAME();
SELECT OrderID FROM dbo.Orders;
GO

EXEC sp_MS_marksystemobject 'dbo.sp_Proc1';
```

Run sp_Proc1 in Northwind, and you will observe that the query ran successfully against the Orders table in Northwind:

```
USE Northwind;
EXEC dbo.sp_Proc1;
```

Make a similar attempt in pubs:

```
USE pubs;
EXEC dbo.sp_Proc1;
master.dbo.sp_Proc1 executing in pubs
Msg 208, Level 16, State 1, Procedure sp_Proc1, Line 5
Invalid object name 'dbo.Orders'.
```

The error tells you that SQL Server looked for an Orders table in pubs but couldn't find one.

When you're done, run the following code for cleanup:

```
USE master;
GO
IF OBJECT_ID('dbo.sp_Proc1') IS NOT NULL
  DROP PROC dbo.sp_Proc1;
GO
USE Northwind
GO
IF OBJECT_ID('dbo.sp_Proc1') IS NOT NULL
  DROP PROC dbo.sp_Proc1;
```

Other Types of Stored Procedures

SQL Server also supports other types of stored procedures:

- **Temporary stored procedures** You can create temporary procedures by prefixing their names with a single number symbol or a double one (# or ##). A single number symbol would make the procedure a local temporary procedure, and two number symbols

would make it a global one. Local and global temporary procedures behave in terms of visibility and scope like local and global temporary tables, respectively.

> **More Info** For details about local and global temporary tables, please refer to Chapter 2.

- **Extended stored procedures** These procedures allow you to create external routines with a programming language such as C using the Open Data Services (ODS) API. These were used in prior versions of SQL Server to extend the functionality of the product. External routines were written using the ODS API, compiled to a .dll file, and registered as extended stored procedures in SQL Server. They were used like user-defined stored procedures with T-SQL. In SQL Server 2005, extended stored procedures are supported for backward compatibility and will be removed in a future version of SQL Server. Now you can rely on the .NET integration in the product and develop CLR stored procedures, as well as other types of routines. I'll cover CLR procedures later in the chapter.

The Stored Procedure Interface

This section covers the interface (that is, the input and output parameters) of stored procedures.

Input Parameters

You can define input parameters for a stored procedure in its header. An input parameter must be provided with a value when the stored procedure is invoked unless you assign the parameter with a default value. As an example, the following code creates the usp_GetCustOrders procedure, which accepts a customer ID and datetime range boundaries as inputs, and returns the given customer's orders in the given datetime range:

```
USE Northwind;
GO

IF OBJECT_ID('dbo.usp_GetCustOrders') IS NOT NULL
  DROP PROC dbo.usp_GetCustOrders;
GO

CREATE PROC dbo.usp_GetCustOrders
  @custid   AS NCHAR(5),
  @fromdate AS DATETIME = '19000101',
  @todate   AS DATETIME = '99991231'
AS

SET NOCOUNT ON;

SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE CustomerID = @custid
  AND OrderDate >= @fromdate
  AND OrderDate < @todate;
GO
```

Tip The SET NOCOUNT ON option tells SQL Server not to produce the message saying how many rows were affected for data manipulation language (DML) statements. Some client database interfaces, such as OLEDB, absorb this message as a row set. The result is that when you expect to get a result set of a query back to the client, instead you get this message of how many rows were affected as the first result set. By issuing SET NOCOUNT ON, you avoid this problem in those interfaces, so you might want to adopt the practice of specifying it.

When invoking a stored procedure, you must specify inputs for those parameters that were not given default values in the definition (for *@custid* in our case). There are two formats for assigning values to parameters when invoking a stored procedure: *unnamed* and *named*. In the unnamed format, you just specify values without specifying the parameter names. Also, you must specify the inputs by declaration order of the parameters. You can omit inputs only for parameters that have default values and that were declared at the end of the parameter list. You cannot omit an input between two parameters for which you do specify values. If you want such parameters to use their default values, you would need to specify the DEFAULT keyword for those.

As an example, the following code invokes the procedure without specifying the inputs for the two last parameters, which will use their default values, and produces the output shown in Table 7-3:

```
EXEC dbo.usp_GetCustOrders N'ALFKI';
```

Table 7-3 Customer ALFKI's Orders

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*
10643	ALFKI	6	1997-08-25 00:00:00.000
10692	ALFKI	4	1997-10-03 00:00:00.000
10702	ALFKI	4	1997-10-13 00:00:00.000
10835	ALFKI	1	1998-01-15 00:00:00.000
10952	ALFKI	1	1998-03-16 00:00:00.000
11011	ALFKI	3	1998-04-09 00:00:00.000

If you want to specify your own value for the third parameter but use the default for the second, specify the DEFAULT keyword for the second parameter:

```
EXEC dbo.usp_GetCustOrders N'ALFKI', DEFAULT, '20060212';
```

This code also produces the output in Table 7-3.

And, of course, if you want to specify your own values for all parameters, just specify them in order, as in:

```
EXEC dbo.usp_GetCustOrders N'ALFKI', '19970101', '19980101';
```

which produces the output shown in Table 7-4:

Table 7-4 Customer ALFKI's Orders in 1997

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*
10643	ALFKI	6	1997-08-25 00:00:00.000
10692	ALFKI	4	1997-10-03 00:00:00.000
10702	ALFKI	4	1997-10-13 00:00:00.000

These are the basics of stored procedures. You're probably already familiar with them, but I decided to include this coverage to lead to a recommended practice. There are many maintenance-related issues that can arise when using the unnamed assignment format. You must specify the arguments in order; you must not omit an optional parameter; and by looking at the code, it might not be clear what the inputs actually mean and to which parameter they relate. Therefore, it's a good practice to use the named assignment format, where you specify the name of the argument and assign it with an input value, as in:

```
EXEC dbo.usp_GetCustOrders
  @custid   = N'ALFKI',
  @fromdate = '19970101',
  @todate   = '19980101';
```

The code is much more readable; you can play with the order in which you specify the inputs; and you can omit any parameter that you like if it has a default value.

Output Parameters

Output parameters allow you to return output values from a stored procedure. A change made to the output parameter within the stored procedure is reflected in the variable from the calling batch that was assigned to the output parameter. The concept is similar to a pointer in C or a *ByRef* parameter in Visual Basic.

As an example, the following code alters the definition of the usp_GetCustOrders procedure, adding to it the output parameter *@numrows*:

```
ALTER PROC dbo.usp_GetCustOrders
  @custid   AS NCHAR(5),
  @fromdate AS DATETIME = '19000101',
  @todate   AS DATETIME = '99991231',
  @numrows  AS INT OUTPUT
AS

SET NOCOUNT ON;
DECLARE @err AS INT;

SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE CustomerID = @custid
  AND OrderDate >= @fromdate
  AND OrderDate < @todate;
```

```
SELECT @numrows = @@rowcount, @err = @@error;

RETURN @err;
GO
```

@numrows will return the number of rows affected by the query. Notice that the stored procedure also uses a RETURN clause to return the value of the *@@error* function after the invocation of the query.

To get the output parameter back from the stored procedure when invoking it, you will need to assign it with a variable defined in the calling batch and mention the keyword OUTPUT. To get back the return status, you will also need to provide a variable from the calling batch right before the procedure name and an equal sign. Here's an example:

```
DECLARE @myerr AS INT, @mynumrows AS INT;

EXEC @myerr = dbo.usp_GetCustOrders
  @custid   = N'ALFKI',
  @fromdate = '19970101',
  @todate   = '19980101',
  @numrows  = @mynumrows OUTPUT;

SELECT @myerr AS err, @mynumrows AS rc;
```

The stored procedure returns the output shown in Table 7-4, plus it assigns the return status *0* to *@myerr* and the number of affected rows (in this case, *3*) to the *@mynumrows* variable.

If you want to manipulate the row set returned by the stored procedure with T-SQL, you will need to create a table first and use the INSERT/EXEC syntax, as shown in Listing 7-2.

Listing 7-2 Send output of *usp_GetCustOrders* to a table

```
IF OBJECT_ID('tempdb..#CustOrders') IS NOT NULL
  DROP TABLE #CustOrders;
GO
CREATE TABLE #CustOrders
(
  OrderID    INT      NOT NULL PRIMARY KEY,
  CustomerID NCHAR(5) NOT NULL,
  EmployeeID INT      NOT NULL,
  OrderDate  DATETIME NOT NULL
);

DECLARE @myerr AS INT, @mynumrows AS INT;

INSERT INTO #CustOrders(OrderID, CustomerID, EmployeeID, OrderDate)
  EXEC @myerr = dbo.usp_GetCustOrders
    @custid   = N'ALFKI',
    @fromdate = '19970101',
    @todate   = '19980101',
    @numrows  = @mynumrows OUTPUT;
```

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM #CustOrders;

SELECT @myerr AS err, @mynumrows AS rc;
GO
```

A client will accept output from a stored procedure into client objects. For example, in ADO programming you define items in a *Parameters* collection for input parameters, output parameters, and return status. A stored procedure can return more than one result set if within it you invoke multiple queries. In the client code, you will absorb the result sets, moving from one record set to another—for example, using the *.NextRecordset* property of the *Recordset* object in ADO. A stored procedure can generate other types of outputs as well, including the output of PRINT and RAISERROR commands. Both would be received by the client through the client interface's structures—for example, the *Errors* collection in ADO.

ADO.NET allows you to accept any possible output from a SQL Server stored procedure at the client side. Of course, in order to accept some output, as the first step, you have to execute the procedure. You execute a stored procedure by using the *SqlCommand* object. To denote you are executing a stored procedure, you have to set the *CommandType* property of the *SqlCommand* object to *CommandType.StoredProcedure*. To define which procedure to execute, you have to insert the name of the procedure to the *CommandText* property. To actually execute the procedure, use the *ExecuteScalar*, *ExecuteNonQuery*, *ExecuteReader*, or *ExecuteXmlReader* methods of the *SqlCommand* object, depending on the output(s) of the stored procedure. Following are the different types of output of stored procedures needed to get the output:

- **A single row set** A single row set can be accepted to an object of the *SqlDataReader* class for the connected environment (connected means that your application maintains a permanent connection to the SQL Server—that is, the connection is always available), and an object of the *SqlDataAdapter* class. If you want to fill an object of the *DataTable* class, which is a member of an object of the *DataSet* class for the disconnected scenario (disconnected here means that after you read the data in the DataTable, in your application, you can disconnect from SQL Server, and you can still use the data read in your application from the DataTable object).
- **Multiple row sets** The *SqlDataReader* class. Use the *NextResult* method of a data reader object to loop through all row sets returned by a stored procedure.
- **Output parameters** Accept output parameters in the *Parameters* collection of a *SqlCommand* object. A *SqlParameter* object in ADO.NET can have four possible directions: *Input*, *Output*, *InputOutput*, or *ReturnValue*. Of course, a single parameter can have a single direction selected at a time. For *ReturnValue* direction, please see the next bullet. Input parameters can be used for input only, and output parameters can be used for output only. SQL Server stored procedure output parameters are actually input/output parameters, so you can pass a value through an output parameter when executing a stored procedure. Therefore, you can specify the *InputOutput* direction of a *SqlParameter* object in ADO.NET, but you have to assign the input value to it before executing the stored procedure or you will get a compile error.

- **Return value** Accept it in a *SqlParameter* object with the *ReturnValue* direction. The return value parameter has to be the first one in the *Parameters* collection of a *SqlCommand* object.
- **Number of rows affected** This can be tricky. You can't rely on the output of SQL Server here, because the developer could add the SET NOCOUNT ON statement to the stored procedure. *SqlDataReader* objects have the *RecordsAffected* property, which gets the number of rows updated, inserted, or deleted. For a SELECT statement, this property can't be used. But there is a problem also with INSERT, UPDATE, and DELETE statements: the *RecordAffected* property gets only the total number of rows affected by all DML statements in the stored procedure. What if you need the number of rows for each DML statement separately? In this case, you can define as many output parameters as the number of DML statements in the procedure, and then store the *@@rowcount* value in every output parameter after every DML statement. This way you can easily get the number of rows affected by SELECT statements as well.
- **Errors** All your .NET code should use a *Try..Catch* block for every risky operation. In the *Catch* block, you can trap real errors—that is, errors with severity levels greater than 10, meaning significant errors, not just warnings or info messages. You run the statements that can produce an error, like executing a stored procedure by using a *SqlCommand* object, in the *Try* block. When an error occurs in the *Try* block, the control of the application is transferred immediately to the *Catch* block, where you can access a *SqlException* object that describes the error. This *SqlException* object has an *Errors* collection. In the collection, you get objects of *SqlError* type, a single object for any error of severity level from 11 through 16 thrown by your SQL Server. You can loop through the collection and read all errors returned by SQL Server. Among the properties of the *SqlError* are a *Number* property, which holds the error number, and a *Message* property, which holds the error message.
- **Warnings** This can be tricky as well. Warnings in SQL Server are error messages with a severity level of 10 or lower. If there is no real error in your code, you can get the warnings in the procedure that handles the *InfoMessage* event of the *SqlConnection* object. The InfoMessage event receives a *SqlInfoMessageEventArgs* object. *SqlInfoMessageEventArgs* has an *Errors* collection, which is similar to previously mentioned Errors collection of the *SQLException* object—it is a collection of objects of *SqlError* type, this time with SQL Server errors of severity level 10 or lower. Again, you can loop through the collection and get all the information from SQL Server warnings that you need. But if there were a real error in the stored procedure, you could catch all the warnings as well as the errors in the *Catch* block, and the *InfoMessage* event would never occur.
- **T-SQL PRINT statement output** You handle this output in the same way that you handle warnings. Read the output using the *InfoMessage* event handler of a *SqlConnection*, or read it in a *Catch* block if there was a real error in the stored procedure.
- **DBCC statement output** Some DBCC commands support the TABLERESULTS option. If you use this option, you can read the output using the *SqlDataReader* object

just as you would read any other row set. If the output of the DBCC statement is textual and not a table, you can get it by using the *InfoMessage* event of the *SqlConnection* object. Again, the same rules apply as for warnings and PRINT output.

- **XML output** ADO.NET 2.0 fully supports the new XML data type, so you can simply use a *SqlDataReader* object to get the results in table format, including XML data type columns. XML output from a SELECT statement with the FOR XML clause can be retrieved into an *XmlReader* object, and you have to use the *ExecuteXmlReader* method of the *SqlCommand* object, of course.
- **User-defined data types (UDTs)** ADO.NET fully supports UDTs as well, so you can fetch values of UDT columns the same way you fetch values of columns of native types. Note that SQL Server sends only the values, not the code for the UDT; therefore, to use any of the UDT's methods at the client side, the code must be available at the client side as well.
- **Schema of a row set retrieved with a *SqlDataReader*** *SqlDataReader* in ADO.NET 2.0 has a new method called *GetSchemaTable*. This method can be used to get a *DataTable* that describes the column metadata of the *SqlDataReader*.

For examples and more details about ADO.NET, please refer to *ADO.NET Examples and Best Practices for C# Programmers* (Apress, 2002) by William R. Vaughn and Peter Blackburn.

When you're done, run the following code for cleanup:

```
USE Northwind;
GO
IF OBJECT_ID('dbo.usp_GetCustOrders') IS NOT NULL
  DROP PROC dbo.usp_GetCustOrders;
GO
IF OBJECT_ID('tempdb..#CustOrders') IS NOT NULL
  DROP TABLE #CustOrders;
GO
```

Resolution

When you create a stored procedure, SQL Server first parses the code to check for syntax errors. If the code passes the parsing stage, successfully, SQL Server attempts to resolve the names it contains. The resolution process verifies the existence of object and column names, among other things. If the referenced objects exist, the resolution process will take place fully—that is, it will also check for the existence of the referenced column names.

If an object name exists but a column within it doesn't, the resolution process will produce an error and the stored procedure will not be created. However, if the object doesn't exist at all, SQL Server will create the stored procedure and defer the resolution process to run time, when the stored procedure is invoked. Of course, if a referenced object or a column is still missing when you execute the stored procedure, the code will fail. This process of postponing name resolution until run time is called *deferred name resolution*.

I'll demonstrate the resolution aspects I just described. First run the following code to make sure that the usp_Proc1 procedure, the usp_Proc2 procedure, and the table T1 do not exist within tempdb:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.usp_Proc1') IS NOT NULL
  DROP PROC dbo.usp_Proc1;
GO
IF OBJECT_ID('dbo.usp_Proc2') IS NOT NULL
  DROP PROC dbo.usp_Proc2;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
```

Run the following code to create the stored procedure usp_Proc1, which refers to a table named T1, which doesn't exist:

```
CREATE PROC dbo.usp_Proc1
AS

SELECT col1 FROM dbo.T1;
GO
```

Because table T1 doesn't exist, resolution was deferred to run time, and the stored procedure was created successfully. If T1 does not exist when you invoke the procedure, it fails at run time. Run the following code:

```
EXEC dbo.usp_Proc1;
```

You will get the following error:

```
Msg 208, Level 16, State 1, Procedure usp_Proc1, Line 6
Invalid object name 'dbo.T1'.
```

Next create table T1 with a column called *col1*:

```
CREATE TABLE dbo.T1(col1 INT);
INSERT INTO dbo.T1(col1) VALUES(1);
```

Invoke the stored procedure again:

```
EXEC dbo.usp_Proc1;
```

This time it will run successfully.

Next, attempt to create a stored procedure called usp_Proc2, referring to a nonexistent column (*col2*) in the existing T1 table:

```
CREATE PROC dbo.usp_Proc2
AS

SELECT col2 FROM dbo.T1;
GO
```

Here, the resolution process was not deferred to run time because T1 exists. The stored procedure was not created, and you got the following error:

```
Msg 207, Level 16, State 1, Procedure usp_Proc2, Line 4
Invalid column name 'col2'.
```

When you're done, run the following code for cleanup:

```
USE tempdb;
GO
IF OBJECT_ID('dbo.usp_Proc1') IS NOT NULL
  DROP PROC dbo.usp_Proc1;
GO
IF OBJECT_ID('dbo.usp_Proc2') IS NOT NULL
  DROP PROC dbo.usp_Proc2;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
```

Compilations, Recompilations, and Reuse of Execution Plans

Earlier I mentioned that when you create a stored procedure, SQL Server parses your code and then attempts to resolve it. If resolution was deferred, it will take place at first invocation. Upon first invocation of the stored procedure, if the resolution phase finished successfully, SQL Server analyzes and optimizes the queries within the stored procedure and generates an execution plan. An execution plan holds the instructions to process the query. These instructions include which order to access the tables in; which indexes, access methods, and join algorithms to use; whether to spool interim sets; and so on. SQL Server typically generates multiple permutations of execution plans and will choose the one with the lowest cost out of the ones that it generated.

Note that SQL Server won't necessarily create all possible permutations of execution plans; if it did, the optimization phase might take too long. SQL Server will limit the optimizer by calculating a threshold for optimization, which is based on the sizes of the tables involved as well as other factors.

Stored procedures can reuse a previously cached execution plan, thereby saving the resources involved in generating a new execution plan. This section will discuss the reuse of execution plans, cases when a plan cannot be reused, and a specific issue relating to plan reuse called the "parameter sniffing problem."

Reuse of Execution Plans

The process of optimization requires mainly CPU resources. SQL Server will, by default, reuse a previously cached plan from an earlier invocation of a stored procedure, without investigating whether it actually is or isn't a good idea to do so.

To demonstrate plan reuse, first run the following code, which creates the usp_GetOrders stored procedure:

```
USE Northwind;
GO
IF OBJECT_ID('dbo.usp_GetOrders') IS NOT NULL
  DROP PROC dbo.usp_GetOrders;
GO

CREATE PROC dbo.usp_GetOrders
  @odate AS DATETIME
AS

SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate >= @odate;
GO
```

The stored procedure accepts an order date as input (*@odate*) and returns orders placed on or after the input order date.

Turn on the STATISTICS IO option to get back I/O information for your session's activity:

```
SET STATISTICS IO ON;
```

Run the stored procedure for the first time, providing an input with *high selectivity* (that is, an input for which a small percentage of rows will be returned); it will generate the output shown in Table 7-5:

```
EXEC dbo.usp_GetOrders '19980506';
```

Table 7-5 Output of EXEC *dbo.usp_GetOrders '19980506'*

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*
11074	SIMOB	7	1998-05-06 00:00:00.000
11075	RICSU	8	1998-05-06 00:00:00.000
11076	BONAP	4	1998-05-06 00:00:00.000
11077	RATTC	1	1998-05-06 00:00:00.000

Examine the execution plan produced for the query, shown in Figure 7-1.

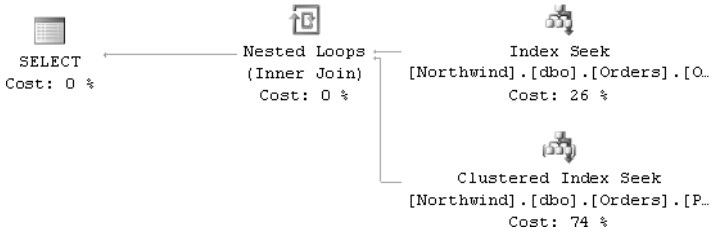

Figure 7-1 Execution plan showing that the index on *OrderDate* is used

Because this is the first time the stored procedure is invoked, SQL Server generated an execution plan for it based on the selective input value and cached that plan.

The optimizer uses cardinality and density information to estimate the cost of the access methods that it considers applying, and the selectivity of filters is an important factor. For example, a query with a highly selective filter can benefit from a nonclustered, noncovering index, while a *low selectivity* filter (that is, one that returns a high percentage of rows) would not justify using such an index.

For highly selective input such as that provided to our stored procedure, the optimizer chose a plan that uses a nonclustered noncovering index on the *OrderDate* column. The plan first performed a seek within that index (Index Seek operator), reaching the first index entry that matches the filter at the leaf level of the index. This seek operation caused two page reads, one at each of the two levels in the index. In a larger table, such an index might contain three or four levels.

Following the seek operation, the plan performed a partial ordered forward scan within the leaf level of the index (which is not seen in the plan but is part of the Index Seek operator). The partial scan fetched all index entries that match the query's filter (that is, all *OrderDate* values greater than or equal to the input *@odate*). Because the input was very selective, only four matching *OrderDate* values were found. In this particular case, the partial scan did not need to access additional pages at the leaf level beyond the leaf page that the seek operation reached, so it did not incur additional I/O.

The plan used a Nested Loops operator, which invoked a series of Clustered Index Seek operations to look up the data row for each of the four index entries that the partial scan found. Because the clustered index on this small table has two levels, the lookups cost eight page reads: *2 × 4 = 8*. In total, there were 10 page reads: *2 (seek) + 2 × 4 (lookups) = 10*. This is the value reported by STATISTICS IO as logical reads.

That's the optimal plan for this selective query with the existing indexes.

Remember that I mentioned earlier that stored procedures will, by default, reuse a previously cached plan? Now that you have a plan stored in cache, additional invocations of the stored procedure will reuse it. That's fine if you keep invoking the stored procedure with a highly selective input. You will enjoy the fact that the plan is reused, and SQL Server will not waste resources on generating new plans. That's especially important with systems that invoke stored procedures very frequently.

However, imagine that the stored procedure's inputs vary considerably in selectivity—some invocations have high selectivity while others have very low selectivity. For example, the following code invokes the stored procedure with an input that has low selectivity:

```
EXEC dbo.usp_GetOrders '19960101';
```

Because there is a plan in cache, it will be reused, which is unfortunate in this case. I provided the minimum *OrderDate* that exists in the table as input. This means that all rows in the table (830) qualify. The plan will require a clustered index lookup for each qualifying row. This invocation generated 1,664 logical reads, even though the whole Orders table resides on 22 data pages. Keep in mind that the Orders table is very small and that in production environments such a table would typically have millions of rows. The cost of reusing such a plan would then be much more dramatic, given a similar scenario. Take a table with 1,000,000 orders, for example, residing on about 25,000 pages. Suppose that the clustered index contains three levels. Just the cost of the lookups would then be 3,000,000 reads: *1,000,000* × *3* = *3,000,000*.

Obviously, in a case such as this, in which a lot of data access is involved and there are large variations in selectivity, it's a very bad idea to reuse a previously cached execution plan.

Similarly, if you invoked the stored procedure for the first time with a low selectivity input, you would get a plan that is optimal for that input—one that issues a table scan (unordered clustered index scan)—and that plan would be cached. Then, in later invocations, the plan would be reused even when the input has high selectivity.

At this point, you can turn off the STATISTICS IO option:

```
SET STATISTICS IO OFF;
```

You can observe the fact that an execution plan was reused by querying the *sys.syscacheobjects* system view (or *master.dbo.syscacheobjects* in SQL Server 2000), which contains information about execution plans:

```
SELECT cacheobjtype, objtype, usecounts, sql
FROM sys.syscacheobjects
WHERE sql NOT LIKE '%cache%'
  AND sql LIKE '%usp_GetOrders%';
```

This query generates the output shown in Table 7-6.

Table 7-6 Execution Plan for usp_GetOrders in sys.syscacheobjects

cacheobjtype	*objtype*	*usecounts*	*sql*
Compiled Plan	Proc	2	CREATE PROC dbo.usp_GetOrders …

Notice that one plan was found for the usp_GetOrders procedure in cache, and that it was used twice (*usecounts* = 2).

One way to solve the problem is to create two stored procedures—one for requests with high selectivity, and a second for low selectivity. You create another stored procedure with flow logic, examining the input and determining which procedure to invoke based on the input's selectivity that your calculations estimate. The idea is nice in theory, but it's very difficult to implement in practice. It can be very complex to calculate the boundary point dynamically

without consuming additional resources. Furthermore, this stored procedure accepts only one input, so imagine how complex things would become with multiple inputs.

Another way to solve the problem is to create (or alter) the stored procedure with the RECOMPILE option, as in:

```
ALTER PROC dbo.usp_GetOrders
  @odate AS DATETIME
WITH RECOMPILE
AS

SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate >= @odate;
GO
```

The RECOMPILE option tells SQL Server to create a new execution plan every time it is invoked. It is especially useful when the time it takes to generate a plan is a small portion of the run time of the stored procedure, and the implications of running the procedure with an inadequate plan would increase the run time substantially.

First run the altered procedure specifying an input with high selectivity:

```
EXEC dbo.usp_GetOrders '19980506';
```

You will get the plan shown in Figure 7-1, which is optimal in this case and generates an I/O cost of 10 logical reads.

Next run it specifying an input with low selectivity:

```
EXEC dbo.usp_GetOrders '19960101';
```

You will get the plan in Figure 7-2, showing a table scan (unordered clustered index scan), which is optimal for this input. The I/O cost in this case is 22 logical reads.

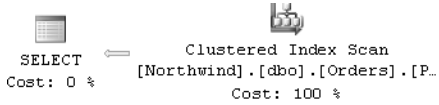

Figure 7-2 Execution plan showing a table scan (unordered clustered index scan)

Note that when creating a stored procedure with the RECOMPILE option, SQL Server doesn't even bother to keep the execution plan for it in cache. If you now query sys.syscacheobjects, you will get no plan back for the usp_GetOrders procedure:

```
SELECT * FROM sys.syscacheobjects
WHERE sql NOT LIKE '%cache%'
  AND sql LIKE '%usp_GetOrders%';
```

In SQL Server 2000, the unit of compilation was the whole stored procedure. So even if you wanted just one particular query to be recompiled, you couldn't request it. If you created the

stored procedure with the RECOMPILE option, the whole procedure went through recompilation every time you invoked it.

SQL Server 2005 supports statement-level recompile. Instead of having all queries in the stored procedure recompiled, SQL Server can now recompile individual statements. You're provided with a new RECOMPILE query hint that allows you to explicitly request a recompilation of a particular query. This way, other queries can benefit from reusing previously cached execution plans if there's no reason to recompile them every time the stored procedure is invoked.

Run the following code to alter the procedure, specifying the RECOMPILE query hint:

```
ALTER PROC dbo.usp_GetOrders
  @odate AS DATETIME
AS

SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate >= @odate
OPTION(RECOMPILE);
GO
```

In our case, there's only one query in the stored procedure, so it doesn't really matter whether you specify the RECOMPILE option at the procedure or the query level. But try to think of the advantages of this hint when you have multiple queries in one stored procedure.

To see that you get good plans, first run the procedure specifying an input with high selectivity:

```
EXEC dbo.usp_GetOrders '19980506';
```

You will get the plan in Figure 7-1, and an I/O cost of 10 logical reads.

Next run it specifying an input with low selectivity:

```
EXEC dbo.usp_GetOrders '19960101';
```

You will get the plan in Figure 7-2 and an I/O cost of 22 logical reads.

Don't get confused by the fact that syscacheobjects shows a plan with the value 2 as the *usecounts*:

```
SELECT cacheobjtype, objtype, usecounts, sql
FROM sys.syscacheobjects
WHERE sql NOT LIKE '%cache%'
  AND sql LIKE '%usp_GetOrders%';
```

The output is the same as in Table 7-6. Remember that if there were other queries in the stored procedure, they could potentially reuse the execution plan.

Recompilations

As I mentioned earlier, a stored procedure will reuse a previously cached execution plan by default. There are exceptions that would trigger a recompilation. Remember that in SQL Server 2000, a recompilation occurs at the whole procedure level, whereas in SQL Server 2005, it occurs at the statement level.

Such exceptions might be caused by issues related to plan correctness or plan optimality. Plan correctness issues include schema changes in underlying objects (for example, adding/dropping a column, adding/dropping an index, and so on) or changes to SET options that can affect query results (for example, ANSI_NULLS, CONCAT_NULL_YIELDS_NULL, and so on). Plan optimality issues that cause recompilation include making data changes in referenced objects to the extent that a new plan might be more optimal—for example, as a result of a statistics update.

Both types of causes for recompilations have many particular cases. At the end of this section, I will provide you with a resource that describes them in great detail.

Naturally, if a plan is removed from cache after a while for lack of reuse, SQL Server will generate a new one when the procedure is invoked again.

To see an example of a cause of a recompilation, first run the following code, which creates the stored procedure usp_CustCities:

```
IF OBJECT_ID('dbo.usp_CustCities') IS NOT NULL
  DROP PROC dbo.usp_CustCities;
GO

CREATE PROC dbo.usp_CustCities
AS

SELECT CustomerID, Country, Region, City,
  Country + '.' + Region + '.' + City AS CRC
FROM dbo.Customers
ORDER BY Country, Region, City;
GO
```

The stored procedure queries the Customers table, concatenating the three parts of the customer's geographical location: *Country*, *Region*, and *City*. By default, the SET option CONCAT_NULL_YIELDS_NULL is turned ON, meaning that when you concatenate a NULL with any string, you get a NULL as a result.

Run the stored procedure for the first time, and you will get the output shown in abbreviated form in Table 7-7:

```
EXEC dbo.usp_CustCities;
```

Table 7-7 Output of *usp_CustCities* when CONCAT_NULL_YIELDS_NULL Is ON (Abbreviated)

CustomerID	*Country*	*Region*	*City*	*CRC*
CACTU	Argentina	NULL	Buenos Aires	NULL
OCEAN	Argentina	NULL	Buenos Aires	NULL
RANCH	Argentina	NULL	Buenos Aires	NULL
ERNSH	Austria	NULL	Graz	NULL
PICCO	Austria	NULL	Salzburg	NULL
MAISD	Belgium	NULL	Bruxelles	NULL
SUPRD	Belgium	NULL	Charleroi	NULL
QUEDE	Brazil	RJ	Rio de Janeiro	Brazil.RJ.Rio de Janeiro
RICAR	Brazil	RJ	Rio de Janeiro	Brazil.RJ.Rio de Janeiro
HANAR	Brazil	RJ	Rio de Janeiro	Brazil.RJ.Rio de Janeiro
GOURL	Brazil	SP	Campinas	Brazil.SP.Campinas
WELLI	Brazil	SP	Resende	Brazil.SP.Resende
TRADH	Brazil	SP	Sao Paulo	Brazil.SP.Sao Paulo
FAMIA	Brazil	SP	Sao Paulo	Brazil.SP.Sao Paulo
COMMI	Brazil	SP	Sao Paulo	Brazil.SP.Sao Paulo
...	...	...	...	...

As you can see, whenever *Region* was NULL, the concatenated string became NULL. SQL Server cached the execution plan of the stored procedure for later reuse. Along with the plan, SQL Server also stored the state of all SET options that can affect query results. You can observe those in a bitmap called *setopts* in sys.syscacheobjects.

Set the CONCAT_NULL_YIELDS_NULL option to OFF, telling SQL Server to treat a NULL in concatenation as an empty string:

```
SET CONCAT_NULL_YIELDS_NULL OFF;
```

And rerun the stored procedure, which will produce the output shown in abbreviated form in Table 7-8:

```
EXEC dbo.usp_CustCities;
```

Table 7-8 Output of *usp_CustCities* when CONCAT_NULL_YIELDS_NULL Is OFF (Abbreviated)

CustomerID	*Country*	*Region*	*City*	*CRC*
CACTU	Argentina	NULL	Buenos Aires	Argentina..Buenos Aires
OCEAN	Argentina	NULL	Buenos Aires	Argentina..Buenos Aires
RANCH	Argentina	NULL	Buenos Aires	Argentina..Buenos Aires
ERNSH	Austria	NULL	Graz	Austria..Graz

Table 7-8 Output of *usp_CustCities* when CONCAT_NULL_YIELDS_NULL Is OFF (Abbreviated)

CustomerID	*Country*	*Region*	*City*	*CRC*
PICCO	Austria	NULL	Salzburg	Austria..Salzburg
MAISD	Belgium	NULL	Bruxelles	Belgium..Bruxelles
SUPRD	Belgium	NULL	Charleroi	Belgium..Charleroi
QUEDE	Brazil	RJ	Rio de Janeiro	Brazil.RJ.Rio de Janeiro
RICAR	Brazil	RJ	Rio de Janeiro	Brazil.RJ.Rio de Janeiro
HANAR	Brazil	RJ	Rio de Janeiro	Brazil.RJ.Rio de Janeiro
GOURL	Brazil	SP	Campinas	Brazil.SP.Campinas
WELLI	Brazil	SP	Resende	Brazil.SP.Resende
TRADH	Brazil	SP	Sao Paulo	Brazil.SP.Sao Paulo
FAMIA	Brazil	SP	Sao Paulo	Brazil.SP.Sao Paulo
COMMI	Brazil	SP	Sao Paulo	Brazil.SP.Sao Paulo
...	...	...	...	...

You can see that when *Region* was NULL, it was treated as an empty string, and as a result, you didn't get a NULL in the *CRC* column. Changing the session option in this case changed the meaning of a query. When you ran this stored procedure, SQL Server first checked whether there was a cached plan that also has the same state of SET options. SQL Server didn't find one, so it had to generate a new plan. Note that regardless of whether the change in the SET option does or doesn't affect the query's meaning, SQL Server looks for a match in the set options state in order to reuse a plan.

Query sys.syscacheobjects, and you will find two plans for usp_CustCities, with two different *setopts* bitmaps, as shown in Table 7-9:

```
SELECT cacheobjtype, objtype, usecounts, setopts, sql
FROM sys.syscacheobjects
WHERE sql NOT LIKE '%cache%'
  AND sql LIKE '%usp_CustCities%';
```

Table 7-9 Execution Plans for *usp_CustCities* in *sys.syscacheobjects*

cacheobjtype	*objtype*	*usecounts*	*setopts*	*sql*
Compiled Plan	Proc	1	4347	CREATE PROC dbo.usp_CustCities ...
Compiled Plan	Proc	1	4339	CREATE PROC dbo.usp_CustCities ...

Why should you care? Client interfaces and tools typically change the state of some SET options whenever you make a new connection to the database. Different client interfaces change different sets of options, yielding different execution environments. If you're using multiple database interfaces and tools to connect to the database and they have different execution environments, they won't be able to reuse each other's plans. You can easily identify the SET options that each client tool changes by running a trace while the applications

connect to the database. If you see discrepancies in the execution environment, you can code explicit SET commands in all applications, which will be submitted whenever a new connection is made. This way, all applications will have sessions with the same execution environment and be able to reuse one another's plans.

When you're done experimenting, turn the CONCAT_NULL_YIELDS_NULL option back ON:

```
SET CONCAT_NULL_YIELDS_NULL ON;
```

This is just one case in which an execution plan is not reused. There are many others. At the end of the following section, I'll provide a resource where you can find more.

Parameter Sniffing Problem

As I mentioned earlier, SQL Server will generate a plan for a stored procedure based on the inputs provided to it upon first invocation, for better or worse. "First invocation" also refers to the first invocation after a plan was removed from cache for lack of reuse or for any other reason. The optimizer "knows" what the values of the input parameters are, and it generates an adequate plan for those inputs. However, things are different when you refer to local variables in your queries. And for the sake of our discussion, it doesn't matter if these are local variables of a plain batch or of a stored procedure. The optimizer cannot "sniff" the content of the variables; therefore, when it optimizes the query, it must make a guess. Obviously, this can lead to poor plans if you're not aware of the problem and don't take corrective measures.

To demonstrate the problem, first insert a new order to the Orders table, specifying the GETDATE function for the *OrderDate* column:

```
INSERT INTO dbo.Orders(OrderDate, CustomerID, EmployeeID)
  VALUES(GETDATE(), N'ALFKI', 1);
```

Alter the usp_GetOrders stored procedure so that it will declare a local variable and use it in the query's filter:

```
ALTER PROC dbo.usp_GetOrders
  @d AS INT = 0
AS

DECLARE @odate AS DATETIME;
SET @odate = DATEADD(day, -@d, CONVERT(VARCHAR(8), GETDATE(), 112));

SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate >= @odate;
GO
```

The procedure defines the integer input parameter *@d* with a default value *0*. It declares a datetime local variable called *@odate*, which is set to today's date minus *@d* days. The stored

procedure then issues a query returning all orders with an *OrderDate* greater than or equal to *@odate*. Invoke the stored procedure using the default value of *@d*, which will generate the output shown in Table 7-10:

```
EXEC dbo.usp_GetOrders;
```

Table 7-10 Output of usp_GetOrders

OrderID	*CustomerID*	*EmployeeID*	*OrderDate*
11079	ALFKI	1	2006-02-12 01:23:53.210

Note The output that you get will have a value in *OrderDate* that reflects the GETDATE value of when you inserted the new order.

The optimizer didn't know what the value of *@odate* was when it optimized the query. So it used a conservative hard-coded value that is 30 percent of the number of rows in the table. For such a low-selectivity estimation, the optimizer naturally chose a table scan, even though the query in practice is highly selective and would be much better off using the index on *OrderDate*.

You can observe the optimizer's estimation and chosen plan by requesting an estimated execution plan (not actual). The estimated execution plan you get for this invocation of the stored procedure is shown in Figure 7-3.

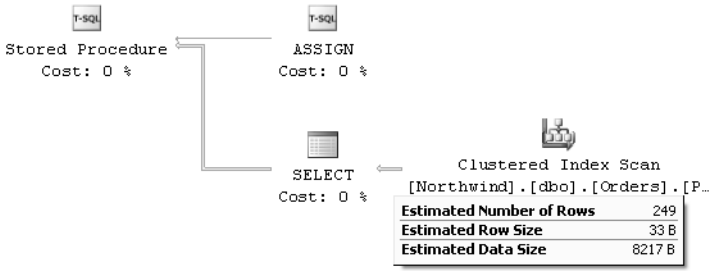

Figure 7-3 Execution plan showing estimated number of rows

You can see that the optimizer chose a table scan (unordered clustered index scan), due to its selectivity estimation of 30 percent (249 rows / 830 total number of rows).

There are several ways to tackle the problem. One is to use, whenever possible, inline expressions in the query that refer to the input parameter instead of a variable. In our case, it is possible:

```
ALTER PROC dbo.usp_GetOrders
  @d AS INT = 0
AS
```

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate >= DATEADD(day, -@d, CONVERT(VARCHAR(8), GETDATE(), 112));
GO
```

Run usp_GetOrders again, and notice the use of the index on *OrderDate* in the execution plan:

```
EXEC dbo.usp_GetOrders;
```

The plan that you will get is similar to the one shown earlier in Figure 7-1. The I/O cost here is just four logical reads.

Another way to deal with the problem is to use a stub procedure. That is, create two procedures. The first procedure accepts the original parameter, assigns the result of the calculation to a local variable, and invokes a second procedure providing it with the variable as input. The second procedure accepts an input order date passed to it and invokes the query that refers directly to the input parameter. When a plan is generated for the procedure that actually invokes the query (the second procedure), the value of the parameter will, in fact, be known at optimization time.

Run the code in Listing 7-3 to implement this solution.

Listing 7-3 Using a stub procedure

```
IF OBJECT_ID('dbo.usp_GetOrdersQuery') IS NOT NULL
  DROP PROC dbo.usp_GetOrdersQuery;
GO

CREATE PROC dbo.usp_GetOrdersQuery
  @odate AS DATETIME
AS

SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate >= @odate;
GO

ALTER PROC dbo.usp_GetOrders
  @d AS INT = 0
AS

DECLARE @odate AS DATETIME;
SET @odate = DATEADD(day, -@d, CONVERT(VARCHAR(8), GETDATE(), 112));

EXEC dbo.usp_GetOrdersQuery @odate;
GO
```

Invoke the usp_GetOrders procedure:

```
EXEC dbo.usp_GetOrders;
```

You will get an optimal plan for the input similar to the one shown earlier in Figure 7-1, yielding an I/O cost of only four logical reads.

Don't forget the issues I described in the previous section regarding the reuse of execution plans. The fact that you got an efficient execution plan for this input doesn't necessarily mean that you would want to reuse it in following invocations. It all depends on whether the inputs are typical or atypical. Make sure you follow the recommendations I gave earlier in case the inputs are atypical.

Finally, there's a new tool provided to you in SQL Server 2005 to tackle the problem—the OPTIMIZE FOR query hint. This hint allows you to provide SQL Server with a literal that reflects the selectivity of the variable, in case the input is typical. For example, if you know that the variable will typically end up with a highly selective value, as you did in our example, you can provide the literal *'99991231'*, which reflects that:

```
ALTER PROC dbo.usp_GetOrders
  @d AS INT = 0
AS

DECLARE @odate AS DATETIME;
SET @odate = DATEADD(day, -@d, CONVERT(VARCHAR(8), GETDATE(), 112));

SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate >= @odate
OPTION(OPTIMIZE FOR(@odate = '99991231'));
GO
```

Run the stored procedure:

```
EXEC dbo.usp_GetOrders;
```

You will get an optimal plan for a highly selective *OrderDate* similar to the one shown earlier in Figure 7-1, yielding an I/O cost of four logical reads.

Note that you might face similar problems when changing the values of input parameters before using them in queries. For example, say you define an input parameter called *@odate* and assign it with a default value of NULL. Before using the parameter in the query's filter, you apply the following code:

```
SET @odate = COALESCE(@odate, '19000101');
```

The query then filters orders where *OrderDate* >= *@odate*. When the query is optimized, the optimizer is not aware of the fact that *@odate* has undergone a change, and it optimizes the query with the original input (NULL) in mind. You will face a similar problem to the one I described with variables, and you should tackle it using similar logic.

More Info For more information on the subject, please refer to the white paper "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005," by Arun Marathe, which can be accessed at *http://www.microsoft.com/technet/prodtechnol/sql/2005/recomp.mspx*.

When you're done, run the following code for cleanup:

```
DELETE FROM dbo.Orders WHERE OrderID > 11077;
GO
IF OBJECT_ID('dbo.usp_GetOrders') IS NOT NULL
  DROP PROC dbo.usp_GetOrders;
GO
IF OBJECT_ID('dbo.usp_CustCities') IS NOT NULL
  DROP PROC dbo.usp_CustCities;
GO
IF OBJECT_ID('dbo.usp_GetOrdersQuery') IS NOT NULL
  DROP PROC dbo.usp_GetOrdersQuery;
GO
```

EXECUTE AS

Stored procedures can play an important security role. You can grant users EXECUTE permissions on the stored procedure without granting them direct access to the underlying objects, thus giving you more control over resource access. However, there are exceptions that would require the caller to have direct permissions on underlying objects. To avoid requiring direct permissions from the caller, all following must be true:

- The stored procedure and the underlying objects belong to the same schema.
- The activity is static (as opposed to using dynamic SQL).
- The activity is DML (SELECT, INSERT, UPDATE, or DELETE), or it is an execution of another stored procedure.

If any listed item is not true, the caller will be required to have direct permissions against the underlying objects. Otherwise, the statements in the stored procedure that do not meet the requirements will fail on a security violation.

That's the behavior in SQL Server 2000, which cannot be changed. That's also the behavior in SQL Server 2005, only now you can set the security context of the stored procedure to that of another user, as if the other user was running the stored procedure. When you create the stored procedure, you can specify an EXECUTE AS clause with one of the following options:

- **CALLER (default)** Security context of the caller
- **SELF** Security context of the user creating or altering the stored procedure
- **OWNER** Security context of the owner of the stored procedure
- **'user_name'** Security context of the specified user name

Remember, all chaining rules and requirements not to have direct permissions for underlying objects still apply, but they apply to the effective user, not the calling user (unless CALLER was specified, of course).

In addition, a user that has impersonation rights can issue an independent EXECUTE AS <option> command to impersonate another entity (login or user). If this is done, it's as if the session changes its security context to that of the impersonated entity.

Parameterizing Sort Order

To practice what you've learned so far, try to provide a solution to the following task: write a stored procedure called usp_GetSortedShippers that accepts a column name from the Shippers table in the Northwind database as one of the inputs (*@colname*), and that returns the rows from the table sorted by the input column name. Assume also that you have a sort direction as input (*@sortdir*), with the value *'A'* representing ascending order and *'D'* representing descending order. The stored procedure should be written with performance in mind—that is, it should use indexes when appropriate (for example, a clustered or nonclustered covering index on the sort column).

Listing 7-4 shows the first suggested solution for the task.

Listing 7-4 Parameterizing sort order, solution 1

```
USE Northwind;
GO
IF OBJECT_ID('dbo.usp_GetSortedShippers') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers;
GO
CREATE PROC dbo.usp_GetSortedShippers
  @colname AS sysname, @sortdir AS CHAR(1) = 'A'
AS

IF @sortdir = 'A'
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY
    CASE @colname
      WHEN N'ShipperID'   THEN CAST(ShipperID AS SQL_VARIANT)
      WHEN N'CompanyName' THEN CAST(CompanyName AS SQL_VARIANT)
      WHEN N'Phone'       THEN CAST(Phone AS SQL_VARIANT)
    END
ELSE
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY
    CASE @colname
      WHEN N'ShipperID'   THEN CAST(ShipperID AS SQL_VARIANT)
      WHEN N'CompanyName' THEN CAST(CompanyName AS SQL_VARIANT)
      WHEN N'Phone'       THEN CAST(Phone AS SQL_VARIANT)
    END DESC;
GO
```

The solution uses an IF statement to determine which of two queries to run based on the requested sort direction. The only difference between the queries is that one uses an ascending

order for the sort expression and the other a descending one. Each query uses a single CASE expression that returns the appropriate column value based on the input column name.

> **Note** SQL Server determines the datatype of the result of a CASE expression based on the datatype with the highest precedence among the possible result values of the expression; not by the datatype of the actual returned value. This means, for example, that if the CASE expression returns a VARCHAR(30) value in one of the THEN clauses and an INT value in another, the result of the expression will always be INT, because INT is higher in precedence than VARCHAR. If in practice the VARCHAR(30) value is returned, SQL Server will attempt to convert it. If the value is not convertible, you get a runtime error. If it is convertible, it becomes an INT and, of course, might have a different sort behavior than the original value.

To avoid such issues, I simply converted all the possible return values to SQL_VARIANT. SQL Server will set the datatype of the CASE expression to SQL_VARIANT, but it will preserve the original base types within that SQL_VARIANT.

Run the following code to test the solution, requesting to sort the shippers by *ShipperID* in descending order, and it will generate the output shown in Table 7-11:

```
EXEC dbo.usp_GetSortedShippers N'ShipperID', N'D';
```

Table 7-11 Output of usp_GetSortedShippers

ShipperID	*CompanyName*	*Phone*
3	Federal Shipping	(503) 555-9931
2	United Package	(503) 555-3199
1	Speedy Express	(503) 555-9831

The output is logically correct, but notice the plan generated for the stored procedure, shown in Figure 7-4.

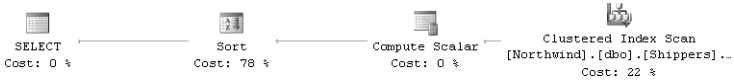

Figure 7-4 Execution plan showing a table scan (unordered clustered index scan) and a sort operator

Remember that the optimizer cannot rely on the sort that the index maintains if you performed manipulation on the sort column. The plan shows a table scan (unordered clustered index scan) followed by an explicit sort operation. For the problem the query was intended to solve, an optimal plan would have performed an ordered scan operation in the clustered index defined on the *ShipperID* column—eliminating the need for an explicit sort operation.

Listing 7-5 shows the second solution for the task.

Listing 7-5 Parameterizing sort order, solution 2

```
ALTER PROC dbo.usp_GetSortedShippers
  @colname AS sysname, @sortdir AS CHAR(1) = 'A'
AS

SELECT ShipperID, CompanyName, Phone
FROM dbo.Shippers
ORDER BY
  CASE WHEN @colname = N'ShipperID'   AND @sortdir = 'A'
    THEN ShipperID   END,
  CASE WHEN @colname = N'CompanyName' AND @sortdir = 'A'
    THEN CompanyName END,
  CASE WHEN @colname = N'Phone'       AND @sortdir = 'A'
    THEN Phone       END,
  CASE WHEN @colname = N'ShipperID'   AND @sortdir = 'D'
    THEN ShipperID   END DESC,
  CASE WHEN @colname = N'CompanyName' AND @sortdir = 'D'
    THEN CompanyName END DESC,
  CASE WHEN @colname = N'Phone'       AND @sortdir = 'D'
    THEN Phone       END DESC;
GO
```

This solution uses CASE expressions in a more sophisticated way. Each column and sort direction combination is treated with its own CASE expression. Only one of the CASE expressions will yield TRUE for all rows, given the column name and sort direction that particular CASE expression is looking for. All other CASE expressions will return NULL for all rows. This means that only one of the CASE expressions—the one that looks for the given column name and sort direction—will affect the order of the output.

Run the following code to test the stored procedure:

```
EXEC dbo.usp_GetSortedShippers N'ShipperID', N'D';
```

Though this stored procedure applies an interesting logical manipulation, it doesn't change the fact that you perform manipulation on the column and don't sort by it as is. This means that you will get a similar nonoptimal plan to the one shown earlier in Figure 7-4.

Listing 7-6 shows the third solution for the task.

Listing 7-6 Parameterizing sort order, solution 3

```
ALTER PROC dbo.usp_GetSortedShippers
  @colname AS sysname, @sortdir AS CHAR(1) = 'A'
AS

IF @colname NOT IN (N'ShipperID', N'CompanyName', N'Phone')
BEGIN
  RAISERROR('Possible SQL injection attempt.', 16, 1);
  RETURN;
END
```

```
DECLARE @sql AS NVARCHAR(4000);

SET @sql = N'SELECT ShipperID, CompanyName, Phone
FROM dbo.Shippers
ORDER BY '
  + QUOTENAME(@colname)
  + CASE @sortdir WHEN 'D' THEN N' DESC' ELSE '' END
  + ';';

EXEC sp_executesql @sql;
GO
```

This solution simply uses dynamic execution, concatenating the input column name and sort direction to the ORDER BY clause of the query. In terms of performance the solution achieves our goal—namely, it will use an index efficiently if an appropriate one exists. To see that it does, run the following code:

```
EXEC dbo.usp_GetSortedShippers N'ShipperID', N'D';
```

Observe in the execution plan shown in Figure 7-5 that the plan performs an ordered backward clustered index scan with no sort operator, which is optimal for these inputs.

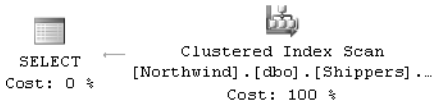

Figure 7-5 Execution plan showing ordered backward clustered index scan

Another advantage of this solution is that it's easy to maintain. The downside of this solution is the use of dynamic execution, which involves many security-related issues (for example, ownership chaining and SQL injection if the inputs are not validated). For details about security issues related to dynamic execution, please refer to Chapter 4.

The fourth solution that I'll cover is shown in Listing 7-7.

Listing 7-7 Parameterizing sort order, solution 4

```
CREATE PROC dbo.usp_GetSortedShippers_ShipperID_A
AS
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY ShipperID;
GO
CREATE PROC dbo.usp_GetSortedShippers_CompanyName_A
AS
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY CompanyName;
GO
CREATE PROC dbo.usp_GetSortedShippers_Phone_A
```

```
AS
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY Phone;
GO
CREATE PROC dbo.usp_GetSortedShippers_ShipperID_D
AS
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY ShipperID   DESC;
GO
CREATE PROC dbo.usp_GetSortedShippers_CompanyName_D
AS
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY CompanyName DESC;
GO
CREATE PROC dbo.usp_GetSortedShippers_Phone_D
AS
  SELECT ShipperID, CompanyName, Phone
  FROM dbo.Shippers
  ORDER BY Phone       DESC;
GO

ALTER PROC dbo.usp_GetSortedShippers
  @colname AS sysname, @sortdir AS CHAR(1) = 'A'
AS

IF @colname = N'ShipperID'        AND @sortdir = 'A'
  EXEC dbo.usp_GetSortedShippers_ShipperID_A;
ELSE IF @colname = N'CompanyName' AND @sortdir = 'A'
  EXEC dbo.usp_GetSortedShippers_CompanyName_A;
ELSE IF @colname = N'Phone'       AND @sortdir = 'A'
  EXEC dbo.usp_GetSortedShippers_Phone_A;
ELSE IF @colname = N'ShipperID'   AND @sortdir = 'D'
  EXEC dbo.usp_GetSortedShippers_ShipperID_D;
ELSE IF @colname = N'CompanyName' AND @sortdir = 'D'
  EXEC dbo.usp_GetSortedShippers_CompanyName_D;
ELSE IF @colname = N'Phone'       AND @sortdir = 'D'
  EXEC dbo.usp_GetSortedShippers_Phone_D;
GO
```

This solution might seem childish at first glance. You create a separate stored procedure with a single static query for each possible combination of inputs. Then, usp_GetSortedShippers can act as a redirector. Simply use a series of IF / ELSE IF statements to check for each possible combination of inputs, and you explicitly invoke the appropriate stored procedure for each. Sure, it is a bit long and requires more maintenance than the previous solution, but it uses static queries that generate optimal plans. Note that each query will get its own plan and will be able to reuse a previously cached plan for the same query.

To test the procedure, run the following code:

```
EXEC dbo.usp_GetSortedShippers N'ShipperID', N'D';
```

You will get the optimal plan for the given inputs, similar to the plan shown earlier in Figure 7-5.

When you're done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.usp_GetSortedShippers') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers;
IF OBJECT_ID('dbo.usp_GetSortedShippers_ShipperID_A') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers_ShipperID_A;
IF OBJECT_ID('dbo.usp_GetSortedShippers_CompanyName_A') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers_CompanyName_A;
IF OBJECT_ID('dbo.usp_GetSortedShippers_Phone_A') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers_Phone_A;
IF OBJECT_ID('dbo.usp_GetSortedShippers_ShipperID_D') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers_ShipperID_D;
IF OBJECT_ID('dbo.usp_GetSortedShippers_CompanyName_D') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers_CompanyName_D;
IF OBJECT_ID('dbo.usp_GetSortedShippers_Phone_D') IS NOT NULL
  DROP PROC dbo.usp_GetSortedShippers_Phone_D;
```

Dynamic Pivot

As another exercise, assume that you're given the task of writing a stored procedure that produces a dynamic pivot in the database you are connected to. The stored procedure accepts the following parameters (all Unicode character strings): *@query*, *@on_rows*, *@on_cols*, *@agg_func* and *@agg_col*. Based on the inputs, you're supposed to construct a PIVOT query string and execute it dynamically. Here's the description of the input parameters:

- ***@query*** Query or table/view name given to the PIVOT operator as input
- ***@on_rows*** Column/expression list that will be used as the grouping columns
- ***@on_cols*** Column or expression to be pivoted; the distinct values from this column will become the target column names
- ***@agg_func*** Aggregate function (MIN, MAX, SUM, COUNT, and so on)
- ***@agg_col*** Column/expression given to the aggregate function as input

If you're still confused regarding the requirements and the meaning of each input, skip the solution in Listing 7-8. Instead, examine the invocation examples and the outputs that follow the listing and the explanation of the solution. Then try to provide your own solution before looking at this one.

Important Note that the solution in Listing 7-8 follows bad programming practices and is insecure. I'll use this solution to discuss flaws in its implementation and then suggest a more robust and secure alternative.

Listing 7-8 shows a suggested solution for the task.

Listing 7-8 Creation script for the sp_pivot stored procedure

```
USE master;
GO

IF OBJECT_ID('dbo.sp_pivot') IS NOT NULL
  DROP PROC dbo.sp_pivot;
GO

CREATE PROC dbo.sp_pivot
  @query    AS NVARCHAR(MAX),
  @on_rows  AS NVARCHAR(MAX),
  @on_cols  AS NVARCHAR(MAX),
  @agg_func AS NVARCHAR(MAX) = N'MAX',
  @agg_col  AS NVARCHAR(MAX)
AS

DECLARE
  @sql     AS NVARCHAR(MAX),
  @cols    AS NVARCHAR(MAX),
  @newline AS NVARCHAR(2);

SET @newline = NCHAR(13) + NCHAR(10);

-- If input is a valid table or view
-- construct a SELECT statement against it
IF COALESCE(OBJECT_ID(@query, N'U'),
            OBJECT_ID(@query, N'V')) IS NOT NULL
  SET @query = N'SELECT * FROM ' + @query;

-- Make the query a derived table
SET @query = N'( ' + @query + @newline + N'      ) AS Query';

-- Handle * input in @agg_col
IF @agg_col = N'*'
  SET @agg_col = N'1';

-- Construct column list
SET @sql =
  N'SET @result = '                                    + @newline +
  N'  STUFF('                                          + @newline +
  N'    (SELECT N'','' + '
          + N'QUOTENAME(pivot_col) AS [text()]'        + @newline +
  N'     FROM (SELECT DISTINCT('
          + @on_cols + N') AS pivot_col'               + @newline +
  N'           FROM' + @query + N') AS DistinctCols'   + @newline +
  N'     ORDER BY pivot_col'                           + @newline +
  N'     FOR XML PATH('''')),'                         + @newline +
  N'    1, 1, N'''');'

EXEC sp_executesql
  @stmt   = @sql,
  @params = N'@result AS NVARCHAR(MAX) OUTPUT',
  @result = @cols OUTPUT;
```

```
-- Create the PIVOT query
SET @sql =
  N'SELECT *'                                          + @newline +
  N'FROM'                                              + @newline +
  N'  ( SELECT '                                       + @newline +
  N'      ' + @on_rows + N','                          + @newline +
  N'      ' + @on_cols + N' AS pivot_col,'             + @newline +
  N'      ' + @agg_col + N' AS agg_col'                + @newline +
  N'    FROM '                                         + @newline +
  N'      ' + @query                                   + @newline +
  N'  ) AS PivotInput'                                 + @newline +
  N'  PIVOT'                                           + @newline +
  N'    ( ' + @agg_func + N'(agg_col)'                 + @newline +
  N'      FOR pivot_col'                               + @newline +
  N'        IN(' + @cols + N')'                        + @newline +
  N'    ) AS PivotOutput;'

EXEC sp_executesql @sql;
GO
```

I'm using this exercise both to explain how to achieve dynamic pivoting and to discuss bad programming practices and security flaws. I'll start by discussing the logic behind the code, and then I'll describe the bad programming practices and flaws and present a more robust and secure solution.

The stored procedure is created as a special procedure in master to allow running it in any database. Remember that dynamic execution is invoked in the context of the current database. This means that the stored procedure's code will effectively run in the context of the current database, interacting with local user objects.

The code checks whether the input parameter *@query* contains a valid table or view. If it does, the code constructs a SELECT statement against the object, storing the statement back in *@query*. If *@query* doesn't contain an existing table/view name, the code assumes that it already contains a query.

The code then makes the query a derived table by adding surrounding parentheses and a derived table alias (*AS Query*). The result string is stored back in *@query*. This derived table will be used both to determine the distinct values that need to be pivoted (from the column/expression stored in the *@on_cols* input parameter) and as the input table expression for the PIVOT operator.

Because the PIVOT operator doesn't support * as an input for the aggregate function—for example, COUNT(*)—the code substitutes a * input in *@agg_col* with the constant *1*.

The code continues by constructing a dynamic query string within the *@sql* variable. This string has code that constructs the column list that will later be served to PIVOT's IN clause. The column list is constructed by a FOR XML PATH query. The query concatenates the distinct list of values from the column/expression stored in the *@on_cols* input parameter.

The concatenation query string (stored in *@sql*) is invoked dynamically. The dynamic code returns through an output parameter a string with the column list, and it assigns it to the variable *@cols*.

The next section of code constructs the actual PIVOT query string in the *@sql* variable. It constructs an outer query against the derived table (aliased as Query), which is currently stored in *@query*. The outer query creates another derived table called PivotInput. The SELECT list in the outer query includes the following items:

- The grouping column/expression list stored in *@on_rows*, which is the part that the PIVOT operator will use in its implicit grouping activity
- The columns/expression to be pivoted (currently stored in *@on_cols*), aliased as *pivot_col*
- The column that will be used as the aggregate function's input (currently stored in *@agg_col*), aliased as *agg_col*

The PIVOT operator works on the derived table PivotInput. Within PIVOT's parentheses, the code embeds the following items: the aggregate function (*@agg_func*) with the aggregate column as its input (*agg_col*), and the column list (*@cols*) within the parentheses of the IN clause. The outermost query simply uses a SELECT * to grab all columns returned from the PIVOT operation.

Finally, the PIVOT query constructed in the *@sql* variable is invoked dynamically.

More Info For in-depth discussion of the PIVOT operator, refer to *Inside T-SQL Querying*.

The sp_pivot stored procedure is extremely flexible, though this flexibility comes at a high security cost, which I'll describe later. To demonstrate its flexibility, I'll provide three examples of invoking it with different inputs. Make sure you study and understand all the inputs carefully.

The following code produces the count of orders per employee and order year, pivoted by order month, and it generates the output shown in Table 7-12:

```
EXEC Northwind.dbo.sp_pivot
  @query    = N'dbo.Orders',
  @on_rows  = N'EmployeeID AS empid, YEAR(OrderDate) AS order_year',
  @on_cols  = N'MONTH(OrderDate)',
  @agg_func = N'COUNT',
  @agg_col  = N'*';
```

Table 7-12 Count of Orders per Employee and Order Year Pivoted by Order Month

empid	*order_year*	1	2	3	4	5	6	7	8	9	10	11	12
1	1996	0	0	0	0	0	0	1	5	5	2	4	9
2	1996	0	0	0	0	0	0	1	2	5	2	2	4
3	1996	0	0	0	0	0	0	4	2	1	3	4	4

Table 7-12 Count of Orders per Employee and Order Year Pivoted by Order Month

empid	*order_year*	1	2	3	4	5	6	7	8	9	10	11	12
4	1996	0	0	0	0	0	0	7	5	3	8	5	3
5	1996	0	0	0	0	0	0	3	0	1	2	2	3
6	1996	0	0	0	0	0	0	2	4	3	0	3	3
7	1996	0	0	0	0	0	0	0	1	2	5	3	0
8	1996	0	0	0	0	0	0	2	6	3	2	2	4
9	1996	0	0	0	0	0	0	2	0	0	2	0	1
1	1997	3	2	5	1	5	4	7	3	8	7	3	7
2	1997	4	1	4	3	3	4	3	1	7	1	5	5
3	1997	7	9	3	5	5	6	2	4	4	7	8	11
4	1997	8	6	4	8	5	5	6	11	5	7	6	10
5	1997	0	0	3	0	2	2	1	3	2	3	1	1
6	1997	2	2	2	4	2	2	2	2	1	4	5	5
7	1997	3	1	2	6	5	1	5	3	5	1	1	3
8	1997	5	8	6	2	4	3	6	5	3	7	2	3
9	1997	1	0	1	2	1	3	1	1	2	1	3	3
1	1998	9	9	11	8	5	0	0	0	0	0	0	0
2	1998	7	3	9	18	2	0	0	0	0	0	0	0
3	1998	10	6	12	10	0	0	0	0	0	0	0	0
4	1998	6	14	12	10	2	0	0	0	0	0	0	0
5	1998	4	6	2	1	0	0	0	0	0	0	0	0
6	1998	3	4	7	5	0	0	0	0	0	0	0	0
7	1998	4	6	4	9	2	0	0	0	0	0	0	0
8	1998	7	2	10	9	3	0	0	0	0	0	0	0
9	1998	5	4	6	4	0	0	0	0	0	0	0	0

The following code produces the sum of the value (quantity * unit price) per employee, pivoted by order year, and it generates the output shown in Table 7-13:

```
EXEC Northwind.dbo.sp_pivot
  @query    = N'
SELECT O.OrderID, EmployeeID, OrderDate, Quantity, UnitPrice
FROM dbo.Orders AS O
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID',
  @on_rows  = N'EmployeeID AS empid',
  @on_cols  = N'YEAR(OrderDate)',
  @agg_func = N'SUM',
  @agg_col  = N'Quantity*UnitPrice';
```

Table 7-13 Sum of Value per Employee Pivoted by Order Year

empid	*1996*	*1997*	*1998*
3	19231.80	111788.61	82030.89
6	17731.10	45992.00	14475.00
9	11365.70	29577.55	42020.75
7	18104.80	66689.14	56502.05
1	38789.00	97533.58	65821.13
4	53114.80	139477.70	57594.95
2	22834.70	74958.60	79955.96
5	21965.20	32595.05	21007.50
8	23161.40	59776.52	50363.11

The following code produces the sum of the quantity per store, pivoted by order year and month, and it generates the output shown in Table 7-14:

```
EXEC pubs.dbo.sp_pivot
  @query    = N'
SELECT stor_id, YEAR(ord_date) AS oy, MONTH(ord_date) AS om, qty
FROM dbo.sales',
  @on_rows  = N'stor_id',
  @on_cols  = N'
CAST(oy AS VARCHAR(4)) + ''_''
  + RIGHT(''0'' + CAST(om AS VARCHAR(2)), 2)',
  @agg_func = N'SUM',
  @agg_col  = N'qty';
```

Table 7-14 Sum of Quantity per Store Pivoted by Order Year and Month

stor_id	*1992_06*	*1993_02*	*1993_03*	*1993_05*	*1993_10*	*1993_12*	*1994_09*
6380	NULL	NULL	NULL	NULL	NULL	NULL	8
7066	NULL	NULL	NULL	50	NULL	NULL	75
7067	80	NULL	NULL	NULL	NULL	NULL	10
7131	NULL	NULL	NULL	85	NULL	NULL	45
7896	NULL	35	NULL	NULL	15	10	NULL
8042	NULL	NULL	25	30	NULL	NULL	25

The implementation of the stored procedure sp_pivot suffers from bad programming practices and security flaws. As I mentioned earlier in the chapter, Microsoft strongly advises against using the *sp_* prefix for user-defined procedure names. On one hand, creating this procedure as a special procedure allows flexibility; on the other hand, by doing so you're relying on behavior that is not supported. It is advisable to forgo the flexibility obtained by creating the procedure with the *sp_* prefix and create it with another prefix as a user-defined stored procedure in the user databases where you need it.

The code defines all input parameters with a virtually unlimited size (using the MAX specifier) and doesn't have any input validation. Because the stored procedure invokes dynamic execution based on user input strings, it's very important to limit the sizes of the inputs and to check those for potential SQL injection attacks. With the existing implementation it's very easy for hackers to inject code that will do havoc and mayhem in your system. You can find discussions about SQL injection in Chapter 4 and in Books Online (URL: *http://msdn2.microsoft.com/en-us/library/ms161953(SQL.90).aspx*). As an example for injecting malicious code through user inputs, consider the following invocation of the stored procedure:

```
EXEC Northwind.dbo.sp_pivot
  @query    = N'dbo.Orders',
  @on_rows  = N'1 AS dummy_col ) DummyTable;
PRINT ''So easy to inject code here!
This could have been a DROP TABLE or xp_cmdshell command!'';
SELECT * FROM (select EmployeeID AS empid',
  @on_cols  = N'MONTH(OrderDate)',
  @agg_func = N'COUNT',
  @agg_col  = N'*';
```

The query string generated by the stored procedure looks like this:

```
SELECT *
FROM
  ( SELECT
      1 AS dummy_col ) DummyTable;
PRINT 'So easy to inject code here!
This could have been a DROP TABLE or xp_cmdshell command!';
SELECT * FROM (select EmployeeID AS empid,
      MONTH(OrderDate) AS pivot_col,
      1 AS agg_col
    FROM
      ( SELECT * FROM dbo.Orders
      ) AS Query
  ) AS PivotInput
  PIVOT
    ( COUNT(agg_col)
      FOR pivot_col
        IN([1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12])
    ) AS PivotOutput;
```

When this code is executed, the injected PRINT statement executes without any problem. I used a harmless PRINT statement just to demonstrate that code can be easily injected here, but obviously the malicious code could be any valid T-SQL code; for example, a DROP TABLE statement, invocation of xp_cmdshell, and so on. In short, it is vital here to take protective measures against SQL injection attempts, as I will demonstrate shortly.

Besides SQL injection attempts, input validation is not performed at all; for example, to verify the validity of input object and column names. The stored procedure also doesn't incorporate exception handling. I discuss exception handling in Chapter 10, so I won't demonstrate it here in the revised solution. I will demonstrate input validation, though.

Before presenting the revised solution, first get rid of the existing sp_pivot implementation:

```
USE master;
GO
IF OBJECT_ID('dbo.sp_pivot') IS NOT NULL
  DROP PROC dbo.sp_pivot;
```

Listing 7-9 shows a suggested revised solution for the task.

Listing 7-9 Creation script for the usp_pivot stored procedure

```
USE Northwind;
GO

IF OBJECT_ID('dbo.usp_pivot') IS NOT NULL
  DROP PROC dbo.usp_pivot;
GO

CREATE PROC dbo.usp_pivot
  @schema_name AS sysname      = N'dbo', -- schema of table/view
  @object_name AS sysname      = NULL,   -- name of table/view
  @on_rows     AS sysname      = NULL,   -- group by column
  @on_cols     AS sysname      = NULL,   -- rotation column
  @agg_func    AS NVARCHAR(12) = N'MAX', -- aggregate function
  @agg_col     AS sysname      = NULL    -- aggregate column
AS

DECLARE
  @object  AS NVARCHAR(600),
  @sql     AS NVARCHAR(MAX),
  @cols    AS NVARCHAR(MAX),
  @newline AS NVARCHAR(2),
  @msg     AS NVARCHAR(500);

SET @newline = NCHAR(13) + NCHAR(10);
SET @object  = QUOTENAME(@schema_name) + N'.' + QUOTENAME(@object_name);

-- Check for missing input
IF   @schema_name IS NULL
  OR @object_name IS NULL
  OR @on_rows     IS NULL
  OR @on_cols     IS NULL
  OR @agg_func    IS NULL
  OR @agg_col     IS NULL
BEGIN
  SET @msg = N'Missing input parameters: '
    + CASE WHEN @schema_name IS NULL THEN N'@schema_name;' ELSE N'' END
    + CASE WHEN @object_name IS NULL THEN N'@object_name;' ELSE N'' END
    + CASE WHEN @on_rows     IS NULL THEN N'@on_rows;'     ELSE N'' END
    + CASE WHEN @on_cols     IS NULL THEN N'@on_cols;'     ELSE N'' END
    + CASE WHEN @agg_func    IS NULL THEN N'@agg_func;'    ELSE N'' END
    + CASE WHEN @agg_col     IS NULL THEN N'@agg_col;'     ELSE N'' END
  RAISERROR(@msg, 16, 1);
  RETURN;
END
```

```
-- Allow only existing table or view name as input object
IF COALESCE(OBJECT_ID(@object, N'U'),
            OBJECT_ID(@object, N'V')) IS NULL
BEGIN
  SET @msg = N'%s is not an existing table or view in the database.';
  RAISERROR(@msg, 16, 1, @object);
  RETURN;
END

-- Verify that column names specified in @on_rows, @on_cols, @agg_col exist
IF   COLUMNPROPERTY(OBJECT_ID(@object), @on_rows, 'ColumnId') IS NULL
  OR COLUMNPROPERTY(OBJECT_ID(@object), @on_cols, 'ColumnId') IS NULL
  OR COLUMNPROPERTY(OBJECT_ID(@object), @agg_col, 'ColumnId') IS NULL
BEGIN
  SET @msg = N'%s, %s and %s must'
    + N' be existing column names in %s.';
  RAISERROR(@msg, 16, 1, @on_rows, @on_cols, @agg_col, @object);
  RETURN;
END

-- Verify that @agg_func is in a known list of functions
-- Add to list as needed and adjust @agg_func size accordingly
IF @agg_func NOT IN
  (N'AVG', N'COUNT', N'COUNT_BIG', N'SUM', N'MIN', N'MAX',
   N'STDEV', N'STDEVP', N'VAR', N'VARP')
BEGIN
  SET @msg = N'%s is an unsupported aggregate function.';
  RAISERROR(@msg, 16, 1, @agg_func);
  RETURN;
END

-- Construct column list
SET @sql =
  N'SET @result = '                                        + @newline +
  N'  STUFF('                                              + @newline +
  N'    (SELECT N'','' + '
           + N'QUOTENAME(pivot_col) AS [text()]'           + @newline +
  N'     FROM (SELECT DISTINCT('
           + QUOTENAME(@on_cols) + N') AS pivot_col'       + @newline +
  N'           FROM ' + @object + N') AS DistinctCols'    + @newline +
  N'     ORDER BY pivot_col'                               + @newline +
  N'     FOR XML PATH('''')),'                             + @newline +
  N'    1, 1, N'''');'

EXEC sp_executesql
  @stmt   = @sql,
  @params = N'@result AS NVARCHAR(MAX) OUTPUT',
  @result = @cols OUTPUT;

-- Check @cols for possible SQL injection attempt
IF   UPPER(@cols) LIKE UPPER(N'%Ox%')
  OR UPPER(@cols) LIKE UPPER(N'%;%')
  OR UPPER(@cols) LIKE UPPER(N'%''%')
  OR UPPER(@cols) LIKE UPPER(N'%--%')
  OR UPPER(@cols) LIKE UPPER(N'%/*%*/%')
```

```
  OR UPPER(@cols) LIKE UPPER(N'%EXEC%')
  OR UPPER(@cols) LIKE UPPER(N'%xp_%')
  OR UPPER(@cols) LIKE UPPER(N'%sp_%')
  OR UPPER(@cols) LIKE UPPER(N'%SELECT%')
  OR UPPER(@cols) LIKE UPPER(N'%INSERT%')
  OR UPPER(@cols) LIKE UPPER(N'%UPDATE%')
  OR UPPER(@cols) LIKE UPPER(N'%DELETE%')
  OR UPPER(@cols) LIKE UPPER(N'%TRUNCATE%')
  OR UPPER(@cols) LIKE UPPER(N'%CREATE%')
  OR UPPER(@cols) LIKE UPPER(N'%ALTER%')
  OR UPPER(@cols) LIKE UPPER(N'%DROP%')
  -- look for other possible strings used in SQL injection here
BEGIN
  SET @msg = N'Possible SQL injection attempt.';
  RAISERROR(@msg, 16, 1);
  RETURN;
END

-- Create the PIVOT query
SET @sql =
  N'SELECT *'                                          + @newline +
  N'FROM'                                              + @newline +
  N'  ( SELECT '                                       + @newline +
  N'      ' + QUOTENAME(@on_rows) + N','               + @newline +
  N'      ' + QUOTENAME(@on_cols) + N' AS pivot_col,'  + @newline +
  N'      ' + QUOTENAME(@agg_col) + N' AS agg_col'     + @newline +
  N'    FROM ' + @object                               + @newline +
  N'  ) AS PivotInput'                                 + @newline +
  N'  PIVOT'                                           + @newline +
  N'    ( ' + @agg_func + N'(agg_col)'                 + @newline +
  N'      FOR pivot_col'                               + @newline +
  N'        IN(' + @cols + N')'                        + @newline +
  N'    ) AS PivotOutput;';

EXEC sp_executesql @sql;
GO
```

This implementation of the stored procedure follows good programming practices and addresses the security flaws mentioned earlier. Keep in mind, however, that when constructing code based on user inputs and stored data/metadata, it is extremely difficult (if at all possible) to achieve complete protection against SQL injection.

The stored procedure usp_pivot is created as a user-defined procedure in the Northwind database with the *usp_* prefix. This means that it isn't as flexible as the previous implementation in the sense that it interacts only with tables and views from Northwind. Note that you can create a view in Northwind that queries objects from other databases, and provide this view as input to the stored procedure.

The usp_pivot stored procedure's code takes several measures to try and prevent SQL injection attempts:

- The sizes of the input parameters are limited.

- Instead of allowing any query as input, the stored procedure accepts only a valid table or view name that exists in the database. Similarly, instead of allowing any T-SQL expression for the arguments *@on_rows*, *@on_cols* and *@agg_col*, the stored procedure accepts only valid column names that exist in the input table/view. Note that you can create a view with any query that you like and serve it as input to the stored procedure.
- The code uses QUOTENAME where relevant to quote object and column names with square brackets.
- The stored procedure's code inspects the *@cols* variable for possible code strings injected to it through data stored in the rotation column values that are being concatenated.

The code also performs input validation to verify that all parameters were supplied; that the table/view and column names exist; and that the aggregate function appears in the list of functions that you want to support. As I mentioned, I discuss exception handling in Chapter 10.

The usp_pivot stored procedure might seem much less flexible than sp_pivot, but remember that you can always create a view to prepare the data for usp_pivot. For example, consider the following code used earlier to return the sum of value (quantity * unit price) per employee, pivoted by order year:

```
EXEC Northwind.dbo.sp_pivot
  @query    = N'
SELECT O.OrderID, EmployeeID, OrderDate, Quantity, UnitPrice
FROM dbo.Orders AS O
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID',
  @on_rows  = N'EmployeeID AS empid',
  @on_cols  = N'YEAR(OrderDate)',
  @agg_func = N'SUM',
  @agg_col  = N'Quantity*UnitPrice';
```

You can achieve the same with usp_pivot by first creating a view that prepares the data:

```
USE Northwind;
GO
IF OBJECT_ID('dbo.ViewForPivot') IS NOT NULL
  DROP VIEW dbo.ViewForPivot;
GO

CREATE VIEW dbo.ViewForPivot
AS

SELECT
  O.OrderID       AS orderid,
  EmployeeID      AS empid,
  YEAR(OrderDate) AS order_year,
  Quantity * UnitPrice AS val
FROM dbo.Orders AS O
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID;
GO
```

Then invoke usp_pivot, as in:

```
EXEC dbo.usp_pivot
  @object_name = N'ViewForPivot',
  @on_rows  = N'empid',
  @on_cols  = N'order_year',
  @agg_func = N'SUM',
  @agg_col  = N'val';
```

You will get the output shown earlier in Table 7-13.

If you think about it, that's a small price to pay compared to compromising the security of your system.

When you're done, run the following code for cleanup:

```
USE Northwind;
GO
IF OBJECT_ID('dbo.ViewForPivot') IS NOT NULL
  DROP VIEW dbo.ViewForPivot;
GO
IF OBJECT_ID('dbo.usp_pivot') IS NOT NULL
  DROP PROC dbo.usp_pivot;
```

CLR Stored Procedures

SQL Server 2005 allows you to develop CLR stored procedures (as well as other routines) using a .NET language of your choice. The previous chapter provided the background about CLR routines, gave advice on when to develop CLR routines versus T-SQL ones, and described the technicalities of how to develop CLR routines. Remember to read Appendix A for instructions on developing, building, deploying, and testing your .NET code. Here I'd just like to give a couple of examples of CLR stored procedures that apply functionality outside the reach of T-SQL code.

The first example is a CLR procedure called usp_GetEnvInfo. This stored procedure collects information from environment variables and returns it in table format. The environment variables that this procedure will return include: Machine Name, Processors, OS Version, CLR Version.

Note that, to collect information from environment variables, the assembly needs external access to operating system resources. By default assemblies are created (using the CREATE ASSEMBLY command) with the most restrictive PERMISSION_SET option – SAFE; meaning that they're limited to accessing database resources only. This is the recommended option to obtain maximum security and stability. The permission set options EXTERNAL_ACCESS and UNSAFE (specified in the CREATE ASSEMBLY or ALTER ASSEMBLY commands, or in the *Project | Properties* dialog in Visual Studio under the *Database* tab) allow external access to system resources such as files, the network, environment variables, or the registry. To allow

EXTERNAL_ACCESS and UNSAFE assemblies to run, you also need to set the database option TRUSTWORTHY to ON. Allowing EXTERNAL_ACCESS or UNSAFE assemblies to run represents a security risk and should be avoided. I will describe a safer alternative shortly, but first I'll demonstrate this option. To set the TRUSTWORTHY option of the CLRUtilities database to ON and to change the permission set of the CLRUtilities assembly to EXTERNAL_ACCESS you would run the following code:

```
-- Database option TRUSTWORTHY needs to be ON for EXTERNAL_ACCESS
ALTER DATABASE CLRUtilities SET TRUSTWORTHY ON;
GO
-- Alter assembly with PERMISSION_SET = EXTERNAL_ACCESS
ALTER ASSEMBLY CLRUtilities
WITH PERMISSION_SET = EXTERNAL_ACCESS;
```

At this point you will be able to run the usp_GetEnvInfo stored procedure. Keep in mind though, that UNSAFE assemblies have complete freedom and can compromise the robustness of SQL Server and the security of the system. EXTERNAL_ACCESS assemblies get the same reliability and stability protection as SAFE assemblies, but from a security perspective they're like UNSAFE assemblies.

A more secure alternative is to sign the assembly with a strong-named key file or Authenticode with a certificate. This strong name (or certificate) is created inside SQL Server as an asymmetric key (or certificate) and has a corresponding login with EXTERNAL ACCESS ASSEMBLY permission (for external access assemblies) or UNSAFE ASSEMBLY permission (for unsafe assemblies). For example, suppose that you have code in the CLRUtilities assembly that needs to run with the EXTERNAL_ACCESS permission set. You can sign the assembly with a strong-named key file from the *Project | Properties* dialog in Visual Studio under the *Signing* tab. Then run the following code to create an asymmetric key from the executable .dll file and a corresponding login with the EXTERNAL_ACCESS ASSEMBLY permission.

```
-- Create an asymmetric key from the signed assembly
-- Note: you have to sign the assembly using a strong name key file
USE master
GO
CREATE ASYMMETRIC KEY CLRUtilitiesKey
  FROM EXECUTABLE FILE =
    'C:\CLRUtilities\CLRUtilities\bin\Debug\CLRUtilities.dll'
-- Create login and grant it with external access permission
CREATE LOGIN CLRUtilitiesLogin FROM ASYMMETRIC KEY CLRUtilitiesKey
GRANT EXTERNAL ACCESS ASSEMBLY TO CLRUtilitiesLogin
GO
```

For more details about securing your assemblies, please refer to Books Online and to the following URL: *http://msdn2.microsoft.com/en-us/library/ms345106.aspx*.

Listing 7-10 shows the definition of the *usp_GetEnvInfo* stored procedure using C# code.

Listing 7-10 CLR usp_GetEnvInfo stored procedure, C# version

```
// Stored procedure that returns environment info in tabular format
[SqlProcedure]
public static void usp_GetEnvInfo()
{
    // Create a record - object representation of a row
    // Include the metadata for the SQL table
    SqlDataRecord record = new SqlDataRecord(
        new SqlMetaData("EnvProperty", SqlDbType.NVarChar, 20),
        new SqlMetaData("Value", SqlDbType.NVarChar, 256));
    // Marks the beginning of the result set to be sent back to the client
    // The record parameter is used to construct the metadata
    // for the result set
    SqlContext.Pipe.SendResultsStart(record);
    // Populate some records and send them through the pipe
    record.SetSqlString(0, @"Machine Name");
    record.SetSqlString(1, Environment.MachineName);
    SqlContext.Pipe.SendResultsRow(record);
    record.SetSqlString(0, @"Processors");
    record.SetSqlString(1, Environment.ProcessorCount.ToString());
    SqlContext.Pipe.SendResultsRow(record);
    record.SetSqlString(0, @"OS Version");
    record.SetSqlString(1, Environment.OSVersion.ToString());
    SqlContext.Pipe.SendResultsRow(record);
    record.SetSqlString(0, @"CLR Version");
    record.SetSqlString(1, Environment.Version.ToString());
    SqlContext.Pipe.SendResultsRow(record);
    // End of result set
    SqlContext.Pipe.SendResultsEnd();
}
```

In this procedure, you can see the usage of some specific extensions to ADO.NET for usage within SQL Server CLR routines. These are defined in the *Microsoft.SqlServer.Server* namespace in .NET 2.0.

When you call a stored procedure from SQL Server, you are already connected. You don't have to open a new connection; you need access to the caller's context from the code running in the server. The caller's context is abstracted in a *SqlContext* object. Before using the *SqlContext* object, you should test whether it is available by using its *IsAvailable* property.

The procedure retrieves some environmental data from the operating system. The data can be retrieved by the properties of an *Environment* object, which can be found in the *System* namespace. But the data you get is in text format. In the CLR procedure, you can see how to generate a row set for any possible format. The routine's code stores data in a *SqlDataRecord* object, which represents a single row of data. It defines the schema for this single row by using the *SqlMetaData* objects.

SELECT statements in a T-SQL stored procedure send the results to the connected caller's "pipe." This is the most effective way of sending results to the caller. The same technique is exposed to CLR routines running in SQL Server. Results can be sent to the connected pipe

using the send methods of the *SqlPipe* object. You can instantiate the *SqlPipe* object with the *Pipe* property of the *SqlContext* object.

Listing 7-11 shows the definition of the usp_GetEnvInfo stored procedure using Visual Basic code.

Listing 7-11 CLR usp_GetEnvInfo stored procedure, Visual Basic version

```
' Stored procedure that returns environment info in tabular format
<SqlProcedure()> _
Public Shared Sub usp_GetEnvInfo()
    ' Create a record - object representation of a row
    ' Include the metadata for the SQL table
    Dim record As New SqlDataRecord( _
        New SqlMetaData("EnvProperty", SqlDbType.NVarChar, 20), _
        New SqlMetaData("Value", SqlDbType.NVarChar, 256))
    ' Marks the beginning of the result set to be sent back to the client
    ' The record parameter is used to construct the metadata for
    ' the result set
    SqlContext.Pipe.SendResultsStart(record)
    '' Populate some records and send them through the pipe
    record.SetSqlString(0, "Machine Name")
    record.SetSqlString(1, Environment.MachineName)
    SqlContext.Pipe.SendResultsRow(record)
    record.SetSqlString(0, "Processors")
    record.SetSqlString(1, Environment.ProcessorCount.ToString())
    SqlContext.Pipe.SendResultsRow(record)
    record.SetSqlString(0, "OS Version")
    record.SetSqlString(1, Environment.OSVersion.ToString())
    SqlContext.Pipe.SendResultsRow(record)
    record.SetSqlString(0, "CLR Version")
    record.SetSqlString(1, Environment.Version.ToString())
    SqlContext.Pipe.SendResultsRow(record)
    ' End of result set
    SqlContext.Pipe.SendResultsEnd()
End Sub
```

Run the following code to register the C# version of the usp_GetEnvInfo stored procedure in the CLRUtilities database:

```
USE CLRUtilities;
GO
IF OBJECT_ID('dbo.usp_GetEnvInfo') IS NOT NULL
  DROP PROC usp_GetEnvInfo;
GO
CREATE PROCEDURE dbo.usp_GetEnvInfo
AS EXTERNAL NAME CLRUtilities.CLRUtilities.usp_GetEnvInfo;
```

Use the following code to register the stored procedure in case you used Visual Basic to develop it:

```
CREATE PROCEDURE dbo.usp_GetEnvInfo
AS EXTERNAL NAME
  CLRUtilities.[CLRUtilities.CLRUtilities].usp_GetEnvInfo;
```

Run the following code to test the usp_GetEnvInfo procedure, generating the output shown in Table 7-15:

```
EXEC dbo.usp_GetEnvInfo;
```

Table 7-15 Output of usp_GetEnvInfo Stored Procedure

EnvProperty	*Value*
Machine Name	DOJO
Processors	1
OS Version	Microsoft Windows NT 5.1.2600 Service Pack 2
CLR Version	2.0.50727.42

The second example for a CLR procedure creates the usp_GetAssemblyInfo stored procedure, which returns information about an input assembly.

Listing 7-12 shows the definition of the usp_GetAssemblyInfo stored procedure using C# code.

Listing 7-12 CLR usp_GetAssemblyInfo stored procedure, C# version

```
// Stored procedure that returns assembly info
// uses Reflection
[SqlProcedure]
public static void usp_GetAssemblyInfo(SqlString asmName)
{
    // Retrieve the clr name of the assembly
    String clrName = null;
    // Get the context
    using (SqlConnection connection =
             new SqlConnection("Context connection = true"))
    {
        connection.Open();
        using (SqlCommand command = new SqlCommand())
        {
            // Get the assembly and load it
            command.Connection = connection;
            command.CommandText =
              "SELECT clr_name FROM sys.assemblies WHERE name = @asmName";
            command.Parameters.Add("@asmName", SqlDbType.NVarChar);
            command.Parameters[0].Value = asmName;
            clrName = (String)command.ExecuteScalar();
            if (clrName == null)
            {
                throw new ArgumentException("Invalid assembly name!");
            }
            Assembly myAsm = Assembly.Load(clrName);
            // Create a record - object representation of a row
            // Include the metadata for the SQL table
            SqlDataRecord record = new SqlDataRecord(
                new SqlMetaData("Type", SqlDbType.NVarChar, 50),
                new SqlMetaData("Name", SqlDbType.NVarChar, 256));
```

```
                // Marks the beginning of the result set to be sent back
                // to the client
                // The record parameter is used to construct the metadata
                // for the result set
                SqlContext.Pipe.SendResultsStart(record);
                // Get all types in the assembly
                Type[] typesArr = myAsm.GetTypes();
                foreach (Type t in typesArr)
                {
                    // Type in a SQL database should be a class or
                    // a structure
                    if (t.IsClass == true)
                    {
                        record.SetSqlString(0, @"Class");
                    }
                    else
                    {
                        record.SetSqlString(0, @"Structure");
                    }
                    record.SetSqlString(1, t.FullName);
                    SqlContext.Pipe.SendResultsRow(record);
                    // Find all public static methods
                    MethodInfo[] miArr = t.GetMethods();
                    foreach (MethodInfo mi in miArr)
                    {
                        if (mi.IsPublic && mi.IsStatic)
                        {
                            record.SetSqlString(0, @"  Method");
                            record.SetSqlString(1, mi.Name);
                            SqlContext.Pipe.SendResultsRow(record);
                        }
                    }
                }
                // End of result set
                SqlContext.Pipe.SendResultsEnd();
            }
        }
    }
```

A DBA could have a problem finding out exactly what part of a particular .NET assembly is loaded to the database. Fortunately, this problem can be easily mitigated. All .NET assemblies include metadata, describing all types (classes and structures) defined within it, including all public methods and properties of the types. In .NET, the *System.Reflection* namespace contains classes and interfaces that provide a managed view of loaded types.

For a very detailed overview of a .NET assembly stored in the file system, you can use the Reflector for .NET, a very sophisticated tool created by Lutz Roeder. Because it is downloadable for free from his site at *http://www.aisto.com/roeder/dotnet/*, it is very popular among .NET developers. Also, Miles Trochesset wrote in his blog at *http://blogs.msdn.com/sqlclr/archive/2005/11/21/495438.aspx* a SQL Server CLR DDL trigger that is fired on the CREATE ASSEMBLY statement. The trigger automatically registers all CLR objects from the assembly, including UDTs, UDAs, UDFs, SPs and triggers. I guess it is going to be very popular among

database developers. I used both tools as a starting point to create my simplified version of a SQL Server CLR stored procedure. I thought that a DBA might prefer to read the assembly metadata from a stored procedure, not from an external tool, like Lutz Roeder's Reflector for .NET is, and also that a DBA might want just to read the metadata first, not immediately to register all CLR objects from the assembly, like Miles Trochesset's trigger does.

The usp_GetAssemblyInfo procedure has to load an assembly from the *sys.assemblies* catalog view. To achieve this task, it has to execute a *SqlCommand*. *SqlCommand* needs a connection. In the usp_GetEnvInfo procedure's code you saw the usage of the *SqlContext* class; now you need an explicit *SqlConnection* object. You can get the context of the caller's connection by using a new connection string option, "*Context connection = true*".

As in the usp_GetEnvInfo procedure, you want to get the results in tabular format. Again you use the *SqlDataRecord* and *SqlMetaData* objects to shape the row returned. Remember that the *SqlPipe* object gives you the best performance to return the row to the caller.

Before you can read the metadata of an assembly, you have to load it. The rest is quite easy. The *GetTypes* method of a loaded assembly can be used to retrieve a collection of all types defined in the assembly. The code retrieves this collection in an array. Then it loops through the array, and for each type it uses the *GetMethods* method to retrieve all public methods in an array of the *MethodInfo* objects. This procedure retrieves type and method names only. The *Reflection* classes allow you to get other metadata information as well—for example, the names and types of input parameters. Listing 7-13 shows the definition of the usp_GetAssemblyInfo stored procedure using Visual Basic code.

Listing 7-13 CLR usp_GetAssemblyInfo stored procedure, Visual Basic version

```
' Stored procedure that returns assembly info
' uses Reflection
<SqlProcedure()> _
Public Shared Sub usp_GetAssemblyInfo(ByVal asmName As SqlString)
    ' Retrieve the clr name of the assembly
    Dim clrName As String = Nothing
    ' Get the context
    Using connection As New SqlConnection("Context connection = true")
        connection.Open()
        Using command As New SqlCommand
            ' Get the assembly and load it
            command.Connection = connection
            command.CommandText = _
              "SELECT clr_name FROM sys.assemblies WHERE name = @asmName"
            command.Parameters.Add("@asmName", SqlDbType.NVarChar)
            command.Parameters(0).Value = asmName
            clrName = CStr(command.ExecuteScalar())
            If (clrName = Nothing) Then
                Throw New ArgumentException("Invalid assembly name!")
            End If
            Dim myAsm As Assembly = Assembly.Load(clrName)
            ' Create a record - object representation of a row
            ' Include the metadata for the SQL table
```

```
                Dim record As New SqlDataRecord( _
                    New SqlMetaData("Type", SqlDbType.NVarChar, 50), _
                    New SqlMetaData("Name", SqlDbType.NVarChar, 256))
                ' Marks the beginning of the result set to be sent back
                ' to the client
                ' The record parameter is used to construct the metadata
                ' for the result set
                SqlContext.Pipe.SendResultsStart(record)
                ' Get all types in the assembly
                Dim typesArr() As Type = myAsm.GetTypes()
                For Each t As Type In typesArr
                    ' Type in a SQL database should be a class or a structure
                    If (t.IsClass = True) Then
                        record.SetSqlString(0, "Class")
                    Else
                        record.SetSqlString(0, "Structure")
                    End If
                    record.SetSqlString(1, t.FullName)
                    SqlContext.Pipe.SendResultsRow(record)
                    ' Find all public static methods
                    Dim miArr() As MethodInfo = t.GetMethods
                    For Each mi As MethodInfo In miArr
                        If (mi.IsPublic And mi.IsStatic) Then
                            record.SetSqlString(0, "  Method")
                            record.SetSqlString(1, mi.Name)
                            SqlContext.Pipe.SendResultsRow(record)
                        End If
                    Next
                Next
                ' End of result set
                SqlContext.Pipe.SendResultsEnd()
            End Using
        End Using
    End Sub
```

Run the following code to register the C# version of the usp_GetAssemblyInfo stored procedure in the CLRUtilities database:

```
IF OBJECT_ID('dbo.usp_GetAssemblyInfo') IS NOT NULL
  DROP PROC usp_GetAssemblyInfo;
GO
CREATE PROCEDURE usp_GetAssemblyInfo
  @asmName AS sysname
AS EXTERNAL NAME CLRUtilities.CLRUtilities.usp_GetAssemblyInfo;
```

And in case you used Visual Basic to develop the stored procedure, use the following code to register it:

```
CREATE PROCEDURE usp_GetAssemblyInfo
  @asmName AS sysname
AS EXTERNAL NAME
  CLRUtilities.[CLRUtilities.CLRUtilities].usp_GetAssemblyInfo;
```

Run the following code to test the usp_GetAssemblyInfo procedure, providing it with the CLRUtilities assembly name as input:

```
EXEC usp_GetAssemblyInfo N'CLRUtilities';
```

You get the output shown in Table 7-16 with the assembly name and the names of all methods (routines) defined within it. You should recognize the routine names except for one—trg_GenericDMLAudit—a CLR trigger that I'll describe in the next chapter.

Table 7-16 Output of usp_GetAssemblyInfo Stored Procedure

Type	*Name*
Class	*CLRUtilities*
Method	*fn_RegExMatch*
Method	*fn_SQLSigCLR*
Method	*fn_ImpCast*
Method	*fn_ExpCast*
Method	*fn_SplitCLR*
Method	*ArrSplitFillRow*
Method	*usp_GetEnvInfo*
Method	*usp_GetAssemblyInfo*
Method	*trg_GenericDMLAudit*

When you're done, run the following code for cleanup:

```
USE CLRUtilities;
GO
IF OBJECT_ID('dbo.usp_GetEnvInfo') IS NOT NULL
  DROP PROC dbo.usp_GetEnvInfo;
GO
IF OBJECT_ID('dbo.usp_GetAssemblyInfo') IS NOT NULL
  DROP PROC dbo.usp_GetAssemblyInfo;
```

Conclusion

Stored procedures are one of the most powerful tools that SQL Server provides you. Understanding them well and using them wisely will result in robust, secure, and well-performing databases. Stored procedures give you a security layer, encapsulation, reduction in network traffic, reuse of execution plans, and much more. SQL Server 2005 introduces the ability to develop CLR routines, eliminating the need to develop extended stored procedures and enhancing the functionality of your database.