Microsoft

Kalen Delaney
Series Editor

2005
EDITION

INSIDE MICROSOFT® SQL SERVER™ 2005: T-SQL PROGRAMMING

Itzik Ben-Gan
Dejan Sarka and Roger Wolter

SQL
Solid Quality Learning

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

ISBN-13: 978-0-7356-2197-8
ISBN-10: 0-7356-2197-7
Library of Congress Control Number 2006924463

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9   QWT   1 0 9 8 7 6

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Acquisitions Editor: Ben Ryan
Project Editor: Kristine Haugseth
Technical Editor: Steve Kass
Indexers: Tony Ross and Lee Ross
Copy Editor: Roger LeBlanc

Body Part No. X11-97537

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 custom aggregate

```
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.

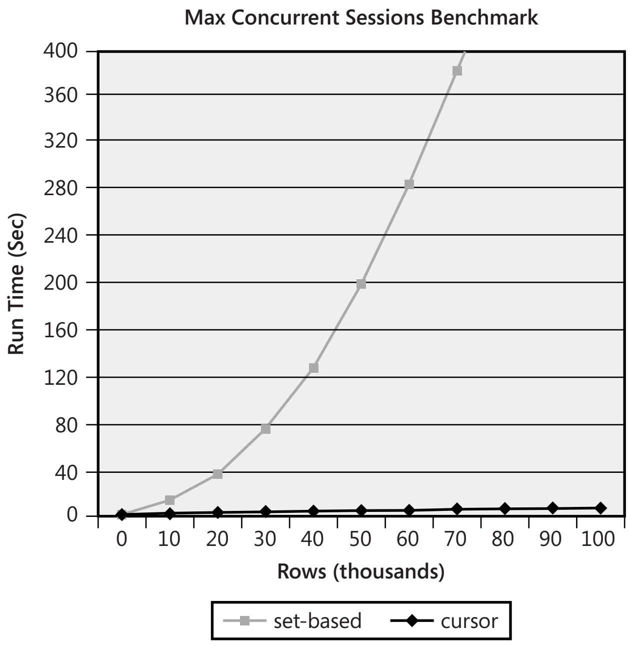

Figure 3-1 Benchmark for running calculations

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 3⅓ 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.

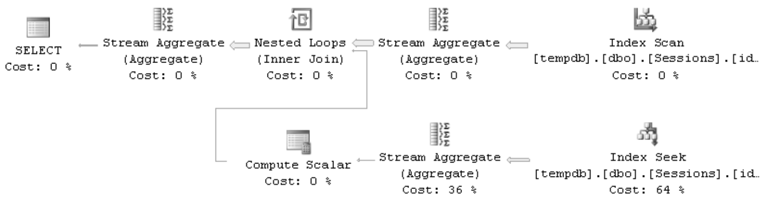

Figure 3-2 Execution plan for Maximum Concurrent Sessions, set-based solution

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)';
```

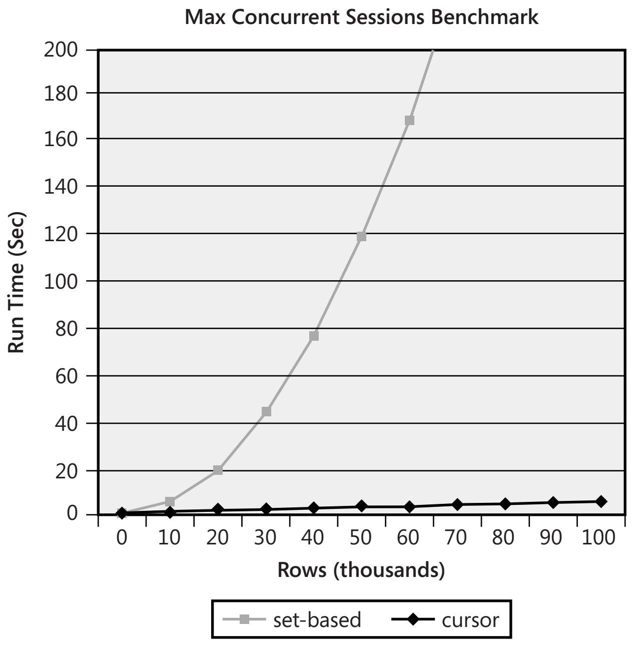

Figure 3-3 Benchmark for Maximum Concurrent Sessions solutions

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(n²)*, 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;
```

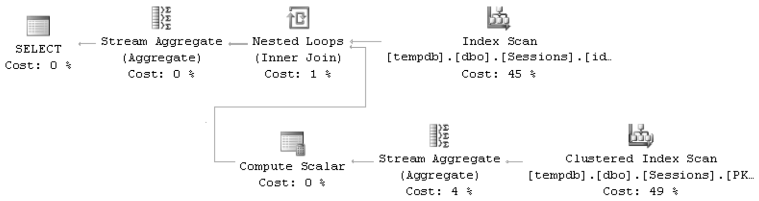

Figure 3-4 Execution plan for revised Maximum Concurrent Sessions solution, small data set

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.

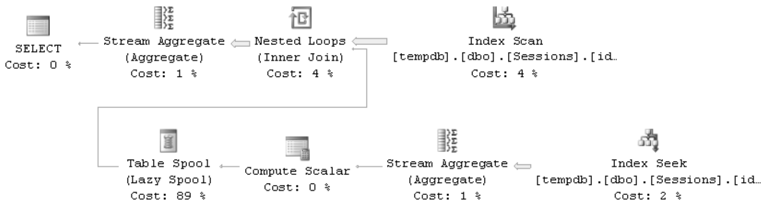

Figure 3-5 Execution plan for revised Maximum Concurrent Sessions solution, large data set with high density

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.

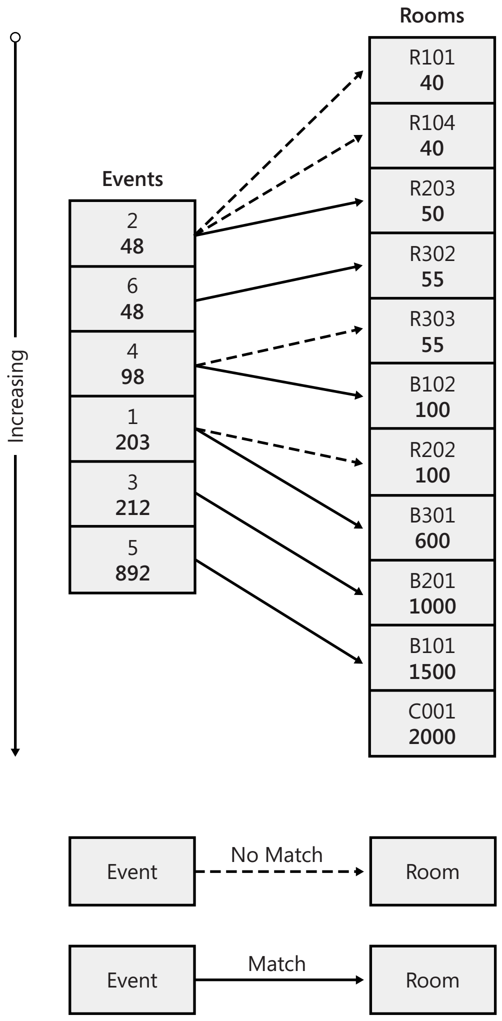

Figure 3-6 Matching algorithm for guaranteed solution scenario

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;
```

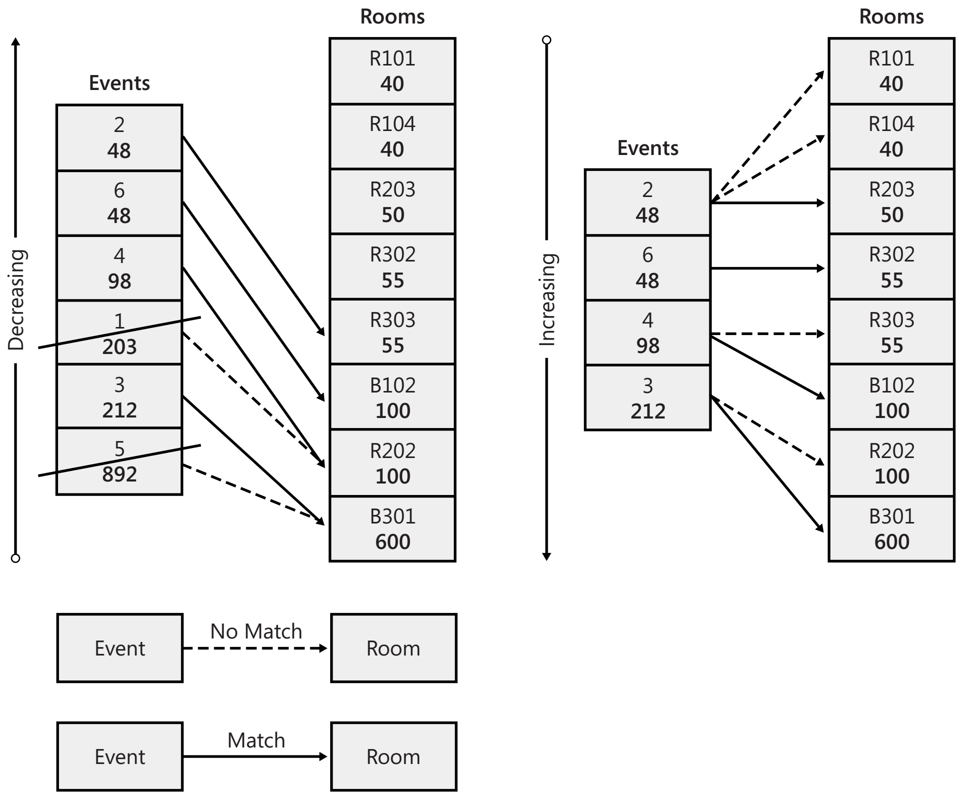

Figure 3-7 Greedy matching algorithm for nonguaranteed solution scenario

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.