Inside Microsoft SQL Server 2005: T-SQL Querying

by Itzik Ben-Gan, Lubor Kollar and Dejan Sarka
Microsoft Press. (c) 2006. Copying Prohibited.

Chapter 3: Query Tuning

Overview

This chapter lays the foundation of query tuning knowledge required for both this book and *Inside Microsoft SQL Server 2005: T-SQL Programming.* (For brevity, I'll refer to the programming book as *Inside T-SQL Programming*, and to both this book and *Inside T-SQL Programming* as "these books.") Here you will be introduced to a tuning methodology, acquire tools for query tuning, learn how to analyze execution plans and perform index tuning, and learn the significance of preparing good sample data and the importance of using set-based solutions.

When building the table of contents for this book, I faced quite a dilemma with regard to the query tuning chapter, a dilemma which I've also faced when teaching advanced T-SQL— should this material appear early or late? On one hand, the chapter provides important background information that is required for the rest of the book; on the other hand, some techniques used for query tuning involve advanced queries—sort of a chicken and egg quandary. I decided to incorporate the chapter early in the book, but I wrote it as an independent unit that can be used as a reference. My recommendation is that you read this chapter before the rest of the book, and when a query uses techniques that you're not familiar with yet, just focus on the conceptual elements described in text. Some queries will use techniques that are described later in the book (for example, pivoting, running aggregations, the OVER clause, CUBE, CTEs, and so on) or in *Inside T-SQL Programming* (for example, temporary tables, cursors, routines, CLR integration, compilations, and so on). Don't be concerned if the techniques are not clear. Feel free, though, to jump to the relevant chapter if you're curious about a certain technique. When you finish reading the books, I suggest that you return to this chapter and revisit any queries that were not clear at first to make sure you fully understand their mechanics.

At the end of this chapter, I'll provide pointers to resources where you can find more information about the subject.

Sample Data for This Chapter

Throughout the chapter, I will use the Performance database and its tables in my examples. Run the code in Listing 3-1 to create the database and its tables, and populate them with sample data. Note that it will take a few minutes for the code to finish.

Listing 3-1: Creation script for sample database and tables

```
SET NOCOUNT ON;
USE master;
GO
IF DB_ID('Performance') IS NULL
  CREATE DATABASE Performance;
GO
USE Performance;
GO

-- Creating and Populating the Nums Auxiliary Table
IF OBJECT_ID('dbo.Nums') IS NOT NULL
  DROP TABLE dbo.Nums;
GO
CREATE TABLE dbo.Nums(n INT NOT NULL PRIMARY KEY);
DECLARE @max AS INT, @rc AS INT;
SET @max = 1000000;
SET @rc = 1;

INSERT INTO Nums VALUES(1);
WHILE @rc * 2 <= @max
BEGIN
  INSERT INTO dbo.Nums SELECT n + @rc FROM dbo.Nums;
  SET @rc = @rc * 2;
END

INSERT INTO dbo.Nums
  SELECT n + @rc FROM dbo.Nums WHERE n + @rc <= @max;
GO

-- Drop Data Tables if Exist
```

```
IF OBJECT_ID('dbo.Orders') IS NOT NULL
  DROP TABLE dbo.Orders;
GO
IF OBJECT_ID('dbo.Customers') IS NOT NULL
  DROP TABLE dbo.Customers;
GO
IF OBJECT_ID('dbo.Employees') IS NOT NULL
  DROP TABLE dbo.Employees;
GO
IF OBJECT_ID('dbo.Shippers') IS NOT NULL
  DROP TABLE dbo.Shippers;
GO

-- Data Distribution Settings
DECLARE
  @numorders   AS INT,
  @numcusts    AS INT,
  @numemps     AS INT,
  @numshippers AS INT,
  @numyears    AS INT,
  @startdate   AS DATETIME;

SELECT
  @numorders   =    1000000,
  @numcusts    =      20000,
  @numemps     =        500,
  @numshippers =          5,
  @numyears    =          4,
  @startdate   = '20030101';

-- Creating and Populating the Customers Table
CREATE TABLE dbo.Customers
(
  custid   CHAR(11)     NOT NULL,
  custname NVARCHAR(50) NOT NULL
);

INSERT INTO dbo.Customers(custid, custname)
  SELECT
    'C' + RIGHT('000000000' + CAST(n AS VARCHAR(10)), 10) AS custid,
    N'Cust_' + CAST(n AS VARCHAR(10)) AS custname
  FROM dbo.Nums
  WHERE n <= @numcusts;

ALTER TABLE dbo.Customers ADD
  CONSTRAINT PK_Customers PRIMARY KEY(custid);

-- Creating and Populating the Employees Table
CREATE TABLE dbo.Employees
(
  empid     INT          NOT NULL,
  firstname NVARCHAR(25) NOT NULL,
  lastname NVARCHAR(25)  NOT NULL
);

INSERT INTO dbo.Employees(empid, firstname, lastname)
  SELECT n AS empid,
    N'Fname_' + CAST(n AS NVARCHAR(10)) AS firstname,
    N'Lname_' + CAST(n AS NVARCHAR(10)) AS lastname
  FROM dbo.Nums
  WHERE n <= @numemps;

ALTER TABLE dbo.Employees ADD
  CONSTRAINT PK_Employees PRIMARY KEY(empid);

-- Creating and Populating the Shippers Table
CREATE TABLE dbo.Shippers
(
  shipperid   VARCHAR(5)   NOT NULL,
  shippername NVARCHAR(50) NOT NULL
);
INSERT INTO dbo.Shippers(shipperid, shippername)
  SELECT shipperid, N'Shipper_' + shipperid AS shippername
  FROM (SELECT CHAR(ASCII('A') - 2 + 2 * n) AS shipperid
        FROM dbo.Nums
        WHERE n <= @numshippers) AS D;

ALTER TABLE dbo.Shippers ADD
  CONSTRAINT PK_Shippers PRIMARY KEY(shipperid);

-- Creating and Populating the Orders Table
CREATE TABLE dbo.Orders
(
  orderid   INT        NOT NULL,
  custid    CHAR(11)   NOT NULL,
  empid     INT        NOT NULL,
  shipperid VARCHAR(5) NOT NULL,
  orderdate DATETIME   NOT NULL,
  filler    CHAR(155)  NOT NULL DEFAULT('a')
);

INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  SELECT n AS orderid,
    'C' + RIGHT('000000000'
            + CAST(
                1 + ABS(CHECKSUM(NEWID())) % @numcusts
                AS VARCHAR(10)), 10) AS custid,
    1 + ABS(CHECKSUM(NEWID())) % @numemps AS empid,
    CHAR(ASCII('A') - 2
           + 2 * (1 + ABS(CHECKSUM(NEWID())) % @numshippers)) AS shipperid,
      DATEADD(day, n / (@numorders / (@numyears * 365.25)), @startdate)
        -- late arrival with earlier date
        - CASE WHEN n % 10 = 0
            THEN THEN 1 + ABS(CHECKSUM(NEWID())) % 30
            ELSE 0
          END AS orderdate
  FROM dbo.Nums
  WHERE n <= @numorders
  ORDER BY CHECKSUM(NEWID());

CREATE CLUSTERED INDEX idx_cl_od ON dbo.Orders(orderdate);

CREATE NONCLUSTERED INDEX idx_nc_sid_od_cid
  ON dbo.Orders(shipperid, orderdate, custid);

CREATE UNIQUE INDEX idx_unc_od_oid_i_cid_eid
  ON dbo.Orders(orderdate, orderid)
  INCLUDE(custid, empid);

ALTER TABLE dbo.Orders ADD
  CONSTRAINT PK_Orders PRIMARY KEY NONCLUSTERED(orderid),
  CONSTRAINT FK_Orders_Customers
    FOREIGN KEY(custid)REFERENCES dbo.Customers(custid),
  CONSTRAINT FK_Orders_Employees
    FOREIGN KEY(empid) REFERENCES dbo.Employees(empid),
  CONSTRAINT FK_Orders_Shippers
    FOREIGN KEY(shipperid) REFERENCES dbo.Shippers(shipperid);
```

The Orders table is the main data table, and it's populated with 1,000,000 orders spanning four years beginning in 2003. The Customers table is populated with 20,000 customers, the Employees table with 500 employees, and the Shippers table with 5 shippers. Note that I distributed the order dates, customer IDs, employee IDs, and shipper IDs in the Orders table with random functions. You might not get the same numbers of rows that I'll be getting in my examples back from the queries, but statistically they should be fairly close.

The Nums table is an auxiliary table of numbers, containing only one column called *n*, populated with integers in the range 1 through 1,000,000.

The code in Listing 3-1 creates the following indexes on the Orders table:

- ***idx_cl_od*** Clustered index on *orderdate*
- ***PK_Orders*** Unique nonclustered index on *orderid*, created implicitly by the primary key
- ***idx_nc_sid_od_cid*** Nonclustered index on *shipperid*, *orderdate*, *custid*
- ***idx_unc_od_oid_i_cid_eid*** Unique nonclustered index on *orderdate*, *orderid*, with included non-key columns *custid*, *empid*

Index structures and their properties will be explained later in the "Index Tuning" section.

Tuning Methodology

This section describes a tuning methodology developed in the company I work for—Solid Quality Learning—and have been implementing with our customers. Credits go to the mentors within the company who took part in developing and practicing the methodology, especially to Andrew J. Kelly, Brian Moran, Fernando G. Guerrero, Eladio Rincón, Dejan Sarka, Mike Hotek, and Ron Talmage, just to name a few.

So, when your system suffers from performance problems, how do you start to solve the problems?

The answer to this question reminds me of a programmer and an IT manager at a company I worked for years ago. The programmer had to finish writing a component and deploy it, but there was a bug in his code that he couldn't find. He produced a printout of the code (which was pretty thick) and went to the IT manager, who was in a meeting. The IT manager was extremely good at detecting bugs, which is why the programmer sought him. The IT manager took the thick printout, opened it, and immediately pointed to a certain line of code. "Here's your bug," he said. "Now go." After the meeting was over, the programmer asked the IT manager how he found the bug so fast? The IT manager replied, "I knew that anywhere I pointed there would be a bug."

Back to query tuning, you can point anywhere in the database and there will be room for tuning there. The question is, is it worth it? For example, would it be worthwhile to tune the concurrency aspects of the system if blocking contributes only to 1 percent of the waits in the system as a whole? It's important to follow a path or methodology that leads you through a series of steps to the main problem areas or bottlenecks in the system—those that contribute to most of the waits. This section will introduce such a methodology.

Before you continue, drop the existing clustered index from the Orders table:

```
USE Performance;
GO
DROP INDEX dbo.Orders.idx_cl_od;
```

Suppose your system suffers from performance problems as a whole—users complain that "everything is slow." Listing 3-2 contains a sampling of queries that run regularly in your system.

Listing 3-2: Sample queries

```
SET NO COUNT ON;
USE Performance;
GO

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 3;

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 5;

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 7;

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
```

```
WHERE orderdate = 20060212';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate = '20060118';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate = '20060828';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060101'
  AND orderdate < '20060201';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060401'
  AND orderdate < '20060501';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20070301';

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060501'
  AND orderdate < '20060601';
```

Restart your SQL Server instance, and then run the code in Listing 3-2 several times (say, 10). SQL Server will internally record performance information you will rely on later. Restarting your instance will reset some of the counters you will rely on later.

When dealing with performance problems, database professionals tend to focus on the technical aspects of the system, such as resource queues, resource utilization, and so on. However, users perceive performance problems simply as waits—they make a request and have to wait to get the results back. A response that takes longer than three seconds to arrive after an interactive request is typically perceived by users as a performance problem. They don't really care how many commands wait on average on each disk spindle or what the cache hit ratio is, and they don't care about blocking, CPU utilization, average page life expectancy in cache, and so on. They care about waits, and that's where performance tuning should start.

The tuning methodology I recommend applies a top-down approach. It starts by investigating waits at the instance level, and then drills down through a series of steps until the processes/ components that generate the bulk of the waits in the system are identified. Once you identify the offending processes, you can focus on tuning them. Following are the main steps of the methodology:

1. Analyze waits at the instance level.
2. Correlate waits with queues.
3. Determine a course of action.
4. Drill down to the database/file level.
5. Drill down to the process level.
6. Tune indexes/queries.

The following sections describe each step in detail.

Analyze Waits at the Instance Level

The first step in the tuning methodology is to identify, at the instance level, which types of waits contribute most to the waits in the system. In SQL Server 2005, you do this by querying a dynamic management view (DMV) called *sys.dm_os_wait_stats*; in SQL Server 2000, you do this by running the command DBCC SQLPERF(WAITSTATS). The aforementioned DMV in SQL Server 2005 is fully documented, and I urge you to read the section describing it in Books Online. I'm not sure why, but the command in SQL Server 2000 is undocumented and surfaced only several years after the product was released. However, from the documentation in SQL Server 2005 you can learn about the different types of waits that are relevant to SQL Server 2000 as well.

The sys.dm_os_wait_stats DMV contains 194 wait types, while the command in SQL Server 2000 will return 77. If you think about it, these are small manageable numbers that are convenient to work with as a starting point. Some other performance tools give you too much information to start with, and create a situation in which you can't see the forest for the trees. I'll continue the discussion assuming you're working with SQL Server 2005.

Run the following query to return the waits in your system sorted by type:

```
SELECT
  wait_type,
  waiting_tasks_count,
  wait_time_ms,
  max_wait_time_ms,
  signal_wait_time_ms
FROM sys.dm_os_wait_stats
ORDER BY wait_type;
```

Table 3-1 shows an abbreviated version of the results I got when I ran this query on my system.

Table 3-1: Contents of *sys.dm_os_wait_stats* in Abbreviated Form

wait_type	waiting_tasks_count	wait_time_ms	max_wait_time_ms	signal_wait_time_
…				
ASYNC_IO_COMPLETION	9	43993	40247	20
ASYNC_NETWORK_IO	69	1682	941	110
CHKPT	1	3234	3234	0
IO_COMPLETION	29167	466270	620	1932
LATCH_EX	41	13589	2082	290
LATCH_SH	3	110	110	30
LATCH_UP	0	0	0	0
LAZYWRITER_SLEEP	284462	266925468	18236	48329
LCK_M_S	3	1882	1612	10
LCK_M_U	41	26898	1992	0
LCK_M_X	0	0	0	0
LOGBUFFER	9495	194920	751	1692
LOGMGR_RESERVE_APPEND	200	198745	1331	50
OLEDB	1451	3034	851	0
PAGELATCH_EX	123	60	20	40
PAGELATCH_SH	25300	365115	2012	54288
PAGELATCH_UP	29	7080	1041	0
PAGEIOLATCH_EX	1259	46536	721	480
PAGEIOLATCH_SH	39491	358205	1762	2733
PAGEIOLATCH_UP	382	6389	480	70
RESOURCE_QUEUE	0	0	0	0
RESOURCE_SEMAPHORE	0	0	0	0
RESOURCE_SEMAPHORE_MUTEX	0	0	0	0
RESOURCE_SEMAPHORE_QUERY_COMPILE	0	0	0	0
RESOURCE_SEMAPHORE_SMALL_QUERY	0	0	0	0

WRITELOG	5176	97400	781	3114
…				

Note Of course, you shouldn't draw conclusions about production systems from the output that I got. Needless to say, my personal machine or your test machine or personal test environment would not necessarily reflect a real production environment. I'm just using the output in this table for illustration purposes. I'll mention later which types of waits we typically find to be predominant in production environments.

The DMV accumulates values since the server was last restarted. If you want to reset its values, run the following code (but don't run it now):

```
DBCC SQLPERF('sys.dm_os_wait_stats', CLEAR);
```

In SQL Server 2000, this code will reset the wait statistics:

```
DBCC SQLPERF(WAITSTATS, CLEAR);
```

The DMV *sys.dm_os_wait_stats* contains the following attributes: *wait_type*; *waiting_tasks_count*, which is the number of waits on this wait type; *wait_time_ms*, which is total wait time for this wait type in milliseconds (including *signal_wait_time*); *max_wait_time_ms*; and *signal_wait_time*, which is the difference between the time the waiting thread was signaled and when it started running.

As I mentioned earlier, you can find information about the various wait types in Books Online. At the end of this chapter, I will point you to resources that provide more detailed information about the different wait types.

Among the various types of waits, you will find ones related to locks, latches (lightweight locks), I/O (including I/O latches), the transaction log, memory, compilations, OLEDB (linked servers and other OLEDB components), and so on. Typically, you will want to ignore some types of waits—for example, those involving a sleep of a thread (meaning that a thread is simply suspended, doing nothing). Make sure you filter out irrelevant waits so that they do not skew your calculations.

In my experience with Solid Quality Learning, I/O waits are by far the most common types of waits our customers need help with. There are several reasons for this. I/O is typically the most expensive resource involved with data-manipulation activities. Also, when queries or indexes are not designed and tuned well, the result is typically excessive I/O. Also, when customers think of "strong" machines, they usually focus their attention on CPU and memory, and they don't pay adequate attention to the I/O subsystem. Database systems need strong I/O subsystems.

Of course, we also deal with other problem areas. There are systems that don't necessarily access large portions of data; instead, these systems involve processes that access small portions of data very frequently. Such is typically the case with online transaction processing (OLTP) environments, which have stored procedures and queries that access small portions of data but are invoked very frequently. In such environments, compilations and recompilations of the code might be the main cause of a bottleneck. OLTP systems also involve a lot of data modification in small portions, and the transaction log often becomes a bottleneck in such environments. The tempdb database can also be a serious bottleneck because all temporary tables, whether created implicitly by an execution plan or explicitly, are created in tempdb. SQL Server also uses tempdb's space to perform other activities. Occasionally, we also find systems with concurrency related (blocking) problems, as well as other problems.

Let's get back to the wait information that you receive from the DMV. You probably won't find it convenient to browse all wait types and try to manually figure out which are the most substantial ones. You want to isolate the top waits—those that in total accumulate to some threshold percentage of the total waits in the system. You can use numbers like 80 percent or 90 percent, because typically a small number of wait types contributes to the bulk of the waits in the system.

The following query isolates the top waits that accumulate in total to 90 percent of the wait time in the system, and it generates (on my system) the output shown in Table 3-2:

Table 3-2: Top Waits

wait_type	wait_time_s	pct	running_pct
IO_COMPLETION	466.24	23.98	23.98
PAGEIOLATCH_SH	365.08	18.78	42.76
ASYNC_NETWORK_IO	358.21	18.42	61.18
LOGMGR_RESERVE_APPEND	198.75	10.22	71.40
LOGBUFFER	194.92	10.02	81.42

WRITELOG	97.40	5.01	86.43
PAGEIOLATCH_EX	46.54	2.39	88.83
ASYNC_IO_COMPLETION	43.99	2.26	91.09

```
WITH Waits AS
(
  SELECT
    wait_type,
    wait_time_ms  /1000. AS wait_time_s,
         100. * wait_time_ms / SUM(wait_time_ms)OVER()ASpct,
         ROW_NUMBER() OVER(ORDER BY wait_time_msDESC) AS rn
      FROM sys.dm_os_wait_stats
      WHERE wait_type NOT LIKE'%SLEEP%'
      --filter out additional irrelevant waits
)
SELECT
  W1.wait_type,
  CAST(W1.wait_time_s AS DECIMAL(12,2))AS wait_time_s,
  CAST(W1.pct AS DECIMAL(12,2))AS pct,
  CAST(SUM(W2.pct)AS DECIMAL(12,2))AS running_pct
FROM Waits AS W1
  JOIN Waits AS W2
    ON W2.rn<= W1.rn
GROUP BY W1.rn, W1.wait_type, W1.wait_time_s, W1.pct
HAVING SUM(W2.pct)-W1.pct < 90-- percentage threshold
ORDER BY W1.rn;
```

This query uses techniques to calculate running aggregates, which I'll explain later in the book. Remember, focus for now on the concepts rather than on the techniques used to achieve them. This query returns the top waits that accumulate to 90 percent of the waits in the system, after filtering out irrelevant wait types. Of course, you can adjust the threshold and filter out other irrelevant waits to your analysis. If you want to see at least *n* rows in the output (say *n = 10*), add the expression *OR W1.rn <=10* to the HAVING clause. With each wait type, the query returns the following: the total wait time in seconds that processes waited on that wait type since the system was last restarted or the counters were cleared; the percentage of the wait time of this type out of the total; and the running percentage from the top-most wait type until the current one.

Note In the *sys.dm_os_wait_stats* DMV, *wait_time_ms* represents the total wait time of all processes that waited on this type, even if multiple processes were waiting concurrently. Still, these numbers would typically give you a good sense of the main problem areas in the system.

Examining the top waits shown in Table 3-2, you can identify three problem areas: I/O, network, and the transaction log. With this information in hand, you are ready for the next step.

I also find it handy to collect wait information in a table and update it at regular intervals (for example, once an hour). By doing this, you can analyze the distribution of waits during the day and identify peak periods.

Run the following code to create the WaitStats table:

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

SELECT GETDATE() AS dt,
  wait_type, waiting_tasks_count, wait_time_ms,
  max_wait_time_ms, signal_wait_time_ms
INTO dbo.WaitStats
FROM sys.dm_os_wait_stats
WHERE 1= 2;

ALTER TABLE dbo.WaitStats
  ADD CONSTRAINT PK_Wait Stats PRIMARY KEY(dt, wait_type);
CREATE INDEX idx_type_dtON dbo.WaitStats(wait_type, dt);
```

Define a job that runs on regular intervals and uses the following code to load the current data from the DMV:

```
INSERT INTO Performance.dbo.Wait Stats
  SELECT GETDATE(),
    wait_type, waiting_tasks_count, wait_time_ms,
    max_wait_time_ms, signal_wait_time_ms
FROM sys.dm_os_wait_stats;
```

Remember that the wait information in the DMV is cumulative. To get the waits that took place within each interval, you need to apply a self-join between two instances of the table— one representing the current samples, and the other representing the previous samples. The join condition will match to each current row the row representing the previous sampling for the same wait type. Then you can subtract the cumulative wait time of the previous sampling from the current, thus producing the wait time during the interval. The following code creates the *fn_interval_waits* function, which implements this logic:

```
IF OBJECT_ID('dbo.fn_interval_waits') IS NOT NULL
  DROP FUNCTION dbo.fn_interval_waits;
GO

CREATE FUNCTION dbo.fn_interval_waits
  (@fromdt AS DATETIME, @todt AS DATETIME)
RETURNS TABLE
AS
RETURN
  WITH Waits AS
  (
    SELECT dt, wait_type, wait_time_ms,
      ROW_NUMBER()OVER(PARTITION BY wait_type
                       ORDER BY dt)ASrn
    FROM dbo.Wait Stats
    WHERE dt>= @fromdt
      AND dt<@todt  +1
  )
   SELECT Prv.wait_type, Prv.dt AS start_time,
     CAST((Cur.wait_time_ms - Prv.wait_time_ms)
            /1000.AS DECIMAL(12, 2))AS interval_wait_s
   FROM Waits AS Cur
     JOIN Waits AS Prv
       ON Cur.wait_type = Prv.wait_type
       AND Cur.rn = Prv.rn + 1
       AND Prv.dt <= @todt;
GO
```

The function accepts the boundaries of a period that you want to analyze. For example, the following query returns the interval waits for the period ‘20060212’ through ‘20060215’ (inclusive), sorted by the totals for each wait type in descending order, wait type, and start time:

```
SELECT wait_type, start_time, interval_wait_s
FROM dbo.fn_interval_waits('20060212', '20060215')AS F
ORDER BY SUM(interval_wait_s) OVER(PARTITION BY wait_type)DESC,
  wait_type, start_time;
```

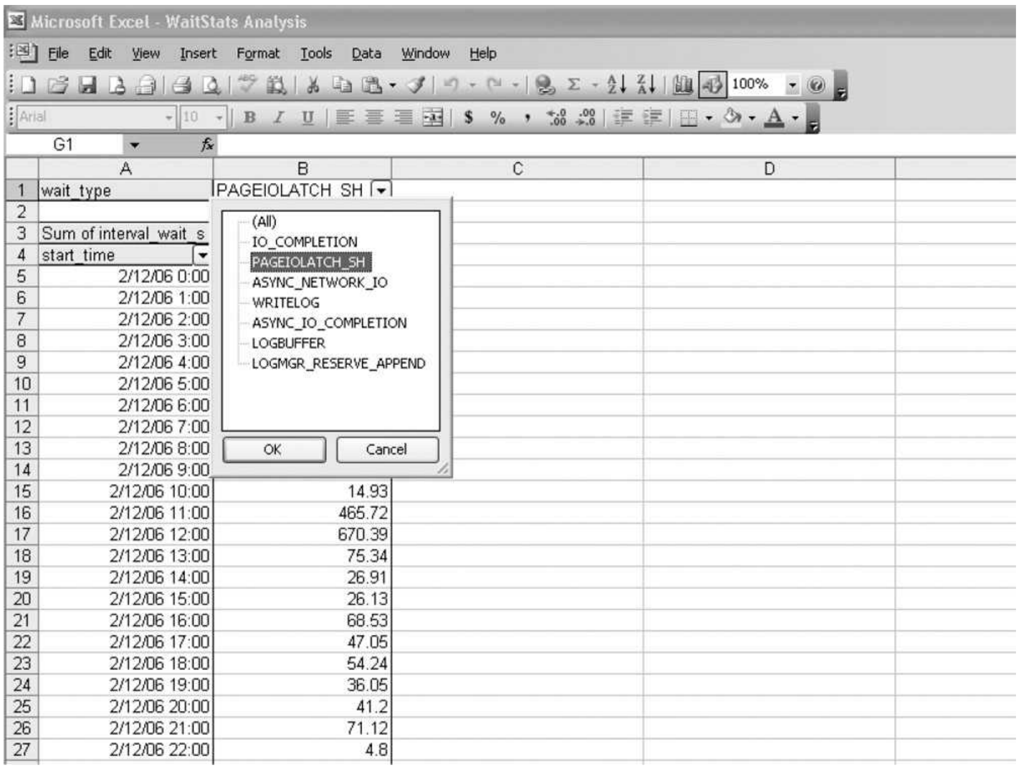
I find Microsoft Office Excel pivot tables or Analysis Services cubes extremely handy in analyzing such information graphically. These tools allow you to easily see the distribution of waits graphically. For example, suppose you want to analyze the waits over the period ‘20060212’ through ‘20060215’ using Excel pivot tables. Prepare the following VIntervalWaits view, which will be used as the external source data for the pivot table:

```
IF OBJECT_ID('dbo.VInterval Waits') IS NOT NULL
  DROP VIEW dbo.VIntervalWaits;
GO

CREATE VIEW dbo.VInterval Waits
AS

SELECT wait_type, start_time, interval_wait_s
FROM dbo.fn_interval_waits('20060212', '20060215')AS F;
GO
```

Create a pivot table and pivot chart in Excel, and specify the *VIntervalWaits* view as the pivot table's external source data. Figure 3-1 shows what the pivot table looks like with my sample data, after filtering only the top waits.



	A	B
1	wait_type	PAGEIOLATCH_SH
2		
3	Sum of interval_wait_s	
4	start_time	
5	2/12/06 0:00	
6	2/12/06 1:00	
7	2/12/06 2:00	
8	2/12/06 3:00	
9	2/12/06 4:00	
10	2/12/06 5:00	
11	2/12/06 6:00	
12	2/12/06 7:00	
13	2/12/06 8:00	
14	2/12/06 9:00	
15	2/12/06 10:00	14.93
16	2/12/06 11:00	465.72
17	2/12/06 12:00	670.39
18	2/12/06 13:00	75.34
19	2/12/06 14:00	26.91
20	2/12/06 15:00	26.13
21	2/12/06 16:00	68.53
22	2/12/06 17:00	47.05
23	2/12/06 18:00	54.24
24	2/12/06 19:00	36.05
25	2/12/06 20:00	41.2
26	2/12/06 21:00	71.12
27	2/12/06 22:00	4.8

Figure 3-1: Pivot table in Excel

Figure 3-2 has a pivot chart, showing graphically the distribution of the PAGEIOLATCH_SH wait type over the input period.

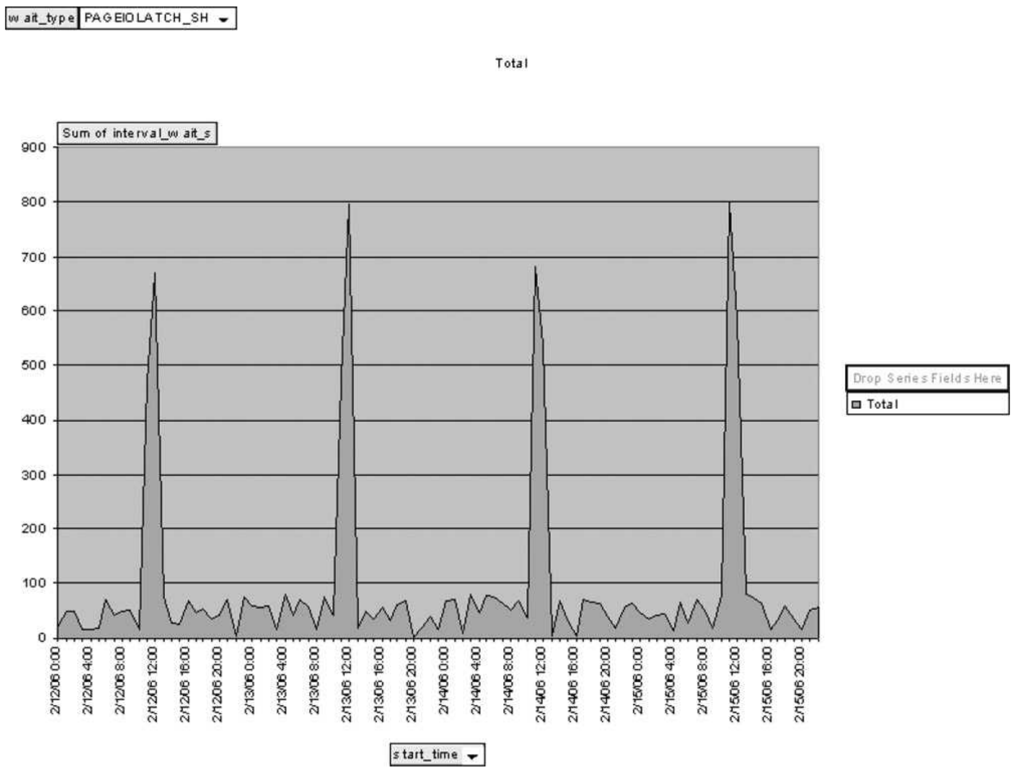


Figure 3-2: Pivot chart 1 in Excel

The PAGEIOLATCH_SH wait type indicates waits on I/O for read operations. You can clearly see that, in our case, there are dramatic peaks every day around noon.

Figure 3-3 has a pivot chart, showing graphically the distribution of all top wait types.

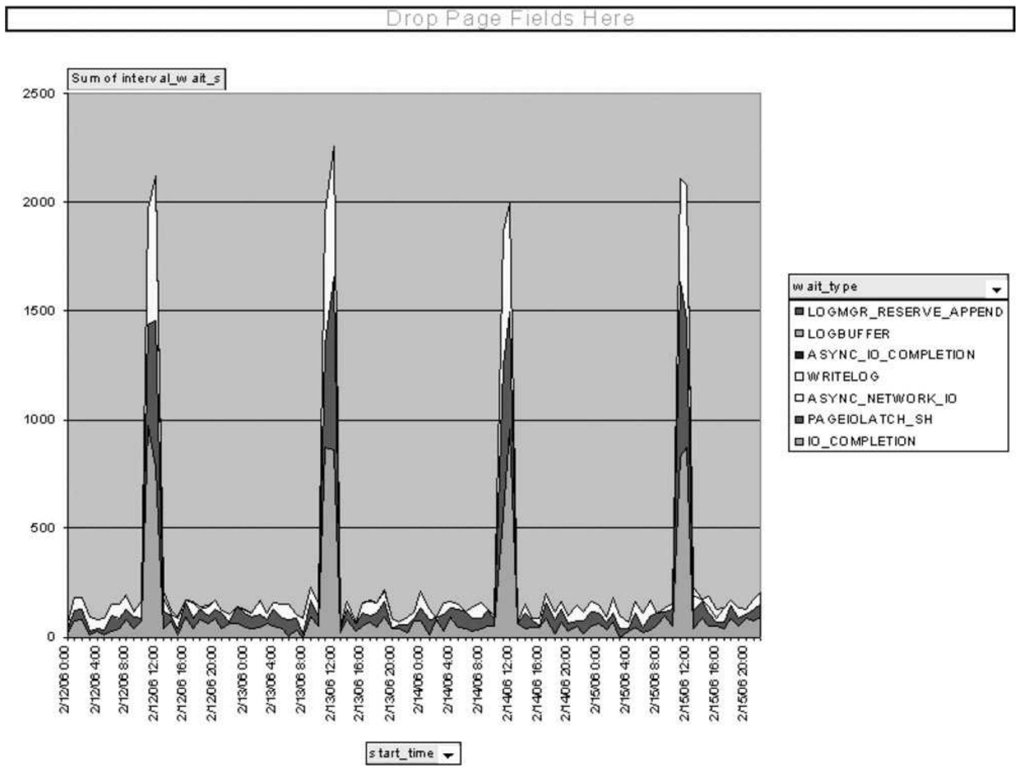


Figure 3-3: Pivot chart 2 in Excel

Again, you can see that most waits occur around noon, daily.

As an example of how handy the analysis of interval waits can be, in one of my tuning projects I found high peaks of I/O latches every four hours that lasted for quite a while (almost the whole four hours), and then dropped. Naturally, in such a case you look for activities that run on a scheduled basis. Sure enough, the "criminal" was isolated. It was a scheduled job that invoked the *sp_updatestats* stored procedure against every database every four hours and ran for almost four hours. This stored procedure is used to update statistics globally at the database level. Statistics are histograms maintained for columns that the optimizer uses to determine selectivity of queries, density of joins, and so on. Apparently, in this case some years prior a query didn't perform well because of a lack of up-to-date statistics on a particular indexed column. The customer got a recommendation back then to refresh statistics, and running the stored procedure seemed to solve the problem. Since then, the customer had been running *sp_updatestats* globally every four hours.

Note that SQL Server automatically creates and updates statistics. Typically, the automatic maintenance of statistics is sufficient, and you should intervene manually only in special cases. And if you do intervene manually, do not use *sp_updatestats* globally! The *sp_updatestats* stored procedure is useful mainly to refresh statistics globally after an upgrade of the product, or after attaching a database from an earlier version of the product or service pack level.

Ironically, when the problem was found, the query that was the trigger for creating the job was not even used anymore in the system. We simply removed the job and let SQL Server use its automatic maintenance of statistics. Naturally, the graph of I/O latches simply flattened and the performance problem was gone.

Correlate Waits with Queues

Once you identify the top waits at the instance level, you should correlate them with queues to identify the problematic resources. You mainly use System Monitor for this task. For example, if you identified I/O-related waits in the previous step, you would check the different I/O queues, cache hit ratios, and memory counters. There should be fewer than two I/O commands waiting on an I/O queue on average per spindle (disk). Cache hit ratios should be as high as possible.

As for memory, it is tightly related to I/O because the more memory you have, the more time pages (data and execution plans) can remain in cache, reducing the need for physical I/O. However, if you have I/O issues, how do you know if adding memory will really help? You need to be familiar with the tools that would help you make the right choice. For example, the counter *SQL Server: Buffer Manager–Page life expectancy* will tell you how many seconds on average a page is expected to remain in cache without reference. Low values indicate that adding memory will allow pages to remain longer in cache, while high values indicate that adding memory won't help you much in this respect. The actual numbers depend on your expectations and the frequency with which you run queries that rely on the same data/execution plans. Typically, numbers greater than several hundred indicate a good state of memory.

But let's say that you have very low values in the counter. Does this mean that you have to add memory? Adding memory in such a case would probably help, but some queries lack important indexes on the source tables and end up performing excessive I/O that could be avoided with a better index design. With less I/O and less memory pressure, the problem can be eliminated without investing in hardware. Of course, if you continue your analysis and realize that your indexes and queries are tuned well, you would then consider hardware upgrades.

Similarly, if you identified other types of waits as the top ones, you would check the relevant queues and resource utilization. For example, if the waits involve compilations/recompilations, you would check the compilations/recompilations and CPU utilization counters, and so on.

SQL Server 2005 provides you with a DMV called *sys.dm_os_performance_counters* containing all the SQL Server object-related counters that you can find in System Monitor. Unfortunately, this DMV doesn't give you the more generic counters, such as CPU utilization, I/O queues, and so on. You have to analyze those externally.

For example, when I ran the following query on my system I got the output shown (in abbreviated form) in Table 3-3:

Table 3-3: Contents of *sys.dm_os_performance_counters* in Abbreviated Form

object_name	counter_name	instance_name	cntr_value	cntr_type
MSSQL$S2: Buffer Manager	Buffercachehitratio		634	537003264
MSSQL$S2: Buffer Manager	Buffer cache hit ratio base		649	1073939712
MSSQL$S2: Buffer Manager	Pagelookups/sec		62882649	272696576
MSSQL$S2: Buffer Manager	Freeliststalls/sec		22370	272696576
MSSQL$S2: Buffer Manager	Freepages		43	65792
MSSQL$S2: Buffer Manager	Totalpages		21330	65792
MSSQL$S2: Buffer Manager	Targetpages		44306	65792
MSSQL$S2: Buffer Manager	Databasepages		19134	65792
MSSQL$S2: Buffer Manager	Reservedpages		0	65792
MSSQL$S2: Buffer Manager	Stolenpages		2153	65792
…				

```
SELECT
  object_name,
  counter_name,
  instance_name,
  cntr_value,
  cntr_type
FROM sys.dm_os_performance_counters;
```

Note that in SQL Server 2000 you query *master.dbo.sysperfinfo* instead.

You might find the ability to query these performance counters in SQL Server useful because you can use query manipulation to analyze the data. As with wait information, you can collect performance counters in a table on regular intervals, and then use queries and tools such as pivot tables to analyze the data over time.

Determine Course of Action

This point—after you have identified the main types of waits and resources involved—is a junction in the tuning process. Based on your discoveries thus far, you will determine a course of action for further investigation. In our case, we need to identify the causes of I/O, networkrelated waits, and transaction log–related waits; then we will continue with a route based on our findings. But if the previous steps had identified blocking problems, compilation/recompilation problems, or others, you would need to proceed with a completely different course of action. Here I'll demonstrate tuning I/O-related performance problems, and at the end of the chapter I'll provide pointers for more information about other related performance problems, some of which are covered in these books.

Drill Down to the Database/File Level

The next step in our tuning process is to drill down to the database/file level. You want to isolate the databases that involve most of the cost. Within the database, you want to drill down to the file type (data/log), as the course of action you take depends on the file type. The tool that allows you to analyze I/O information at the database/file level is a dynamic management function (DMF) called *sys.dm_io_virtual_file_stats.* The function accepts a database ID and file ID as inputs, and returns I/O information about the input database file. You specify NULLs in both to request information about all databases and all files. Note that in SQL Server 2000 you query a function called: *fn_virtualfilestats*, providing it–1 in both parameters to get information about all databases.

The function returns the attributes: *database_id*; *file_id*; *sample_ms*, which is the number of milliseconds since the instance of SQL Server has started (and can be used to compare different outputs from this function); *num_of_reads*; *num_of_bytes_read*; *io_stall_read_ms*, which is the total time, in milliseconds, that the users waited for reads issued on the file; *num_of_writes*; *num_of_bytes_written*; *io_stall_write_ms*; *io_stall*, which is the total time, in milliseconds, that users waited for I/O to be completed on the file; *size_on_disk_bytes* (in bytes); and *file_handle*, which is the Microsoft Windows file handle for this file.

Note The measurements are reset when SQL Server starts, and they only indicate physical I/O against the files and not logical I/O.

At this point, we want to figure out which databases involve most of the I/O and I/O stalls in the system and, within the database, which file type (data/log). The following query will give you this information, sorted in descending order by the I/O stalls, and it will generate (on my system) the output shown in abbreviated form in Table 3-4:

Table 3-4: Database I/O Information in Abbreviated Form

db	file_type	io_mb	io_stall_s	io_stall_pct	rn
Performance	data	11400.81	11172.55	74.86	1
Performance	log	3732.40	2352.55	15.76	2
tempdb	data	940.08	1327.59	8.90	3
Generic	data	54.10	21.78	0.15	4
master	log	7.33	13.75	0.09	5
tempdb	log	5.24	11.20	0.08	6
master	data	13.56	6.20	0.04	7
Generic	log	1.02	3.81	0.03	8
msdb	data	5.07	2.76	0.02	9
AdventureWorks	log	0.55	1.71	0.01	10
…					

```
WITH DBIO AS
(
 SELECT
   DB_NAME(IVFS.database_id) AS db,
   CASE WHEN MF.type = 1 THEN 'log' ELSE 'data' END AS file_type,
   SUM(IVFS.num_of_bytes_read +IVFS.num_of_bytes_written) AS io,
   SUM(IVFS.io_stall) AS io_stall
 FROM sys.dm_io_virtual_file_stats(NULL, NULL) AS IVFS
   JOIN sys.master_files AS MF
     ON IVFS.database_id = MF.database_id
     AND IVFS.file_id = MF.file_id
 GROUP BY DB_NAME(IVFS.database_id), MF.type
)
SELECT db, file_type,
  CAST(1. *io/ (1024 *1024) AS DECIMAL(12, 2))AS io_mb,
  CAST(io_stall /1000. AS DECIMAL(12,2))ASio_stall_s,
  CAST(100.*io_stall / SUM(io_stall)OVER()
       AS DECIMAL(10,2))ASio_stall_pct,
  ROW_NUMBER()OVER(ORDER BY io_stall DESC) AS rn
FROM DBIO
ORDER BY io_stall DESC;
```

The output shows the database name, file type, total I/O (reads and writes) in megabytes, I/O stalls in seconds, I/O stalls in percent of the total for the whole system, and a row number indicating a position in the sorted list based on I/O stalls. Of course, if you want, you can calculate a percentage and row number based on I/O as opposed to I/O stalls, and you can also use running aggregation techniques to calculate a running percentage, as I demonstrated earlier. You might also be interested in a separation between the reads and writes for your analysis. In this output, you can clearly identify the three main elements involving most of the system's I/O stalls—the data portion of Performance, which scores big time; the log portion of Performance; and the data portion of tempdb. Obviously, you should focus on these three elements, with special attention to data activity against the Performance database.

Regarding the bulk of our problem—I/O against the data portion of the Performance database—you now need to drill down to the process level to identify the processes that involve most of the waits.

As for the transaction log, you first need to check whether it's configured adequately. That is, whether it is placed on its own disk drive with no interference, and if so, whether the disk drive is fast enough. If the log happens to be placed on a slow disk drive, you might want to consider dedicating a faster disk for it. Once the logging activity exceeds the throughput of the disk drive, you start getting waits and stalls. You might be happy with dedicating a faster disk drive for the log, but then again, you might not have the budget or you might have already assigned the fastest disk you could for it. Keep in mind that the transaction log is written sequentially, so striping it over multiple disk drives won't help, unless you also have activities that read from the log (such as transaction log replication, or triggers in SQL Server 2000). You might also be able to optimize the processes that cause intensive logging by reducing their amount of logging. I'll elaborate on the transaction log and its optimization in Chapter 8.

As for tempdb, there are many activities—both explicit and implicit—that might cause tension in tempdb to the point where it can become a serious bottleneck in the system. In fact, internally tempdb is used more heavily in SQL Server 2005 because of the new row-versioning technology incorporated in the engine. Row versioning is used for the new snapshot-based isolations, the special inserted and deleted tables in triggers, the new online index operations, and the new multiple active result sets (MARS). The new technology maintains an older, consistent version of a row in a linked list in tempdb. Typically, there's a lot of room for optimizing tempdb, and you should definitely give that option adequate attention. I'll elaborate on tempdb and on row versioning in *Inside T-SQL Programming* in the chapters that cover temporary tables, triggers, and transactions.

For our demonstration, let's focus on solving the I/O problems related to the data portion of the Performance database.

Drill Down to the Process Level

Now that you know which databases (in our case, one) involve most of the performance problem, you want to drill down to the process level; namely, identify the processes (stored procedures, queries, and so on) that need to be tuned. For this task, you will find SQL Server's built-in tracing capabilities extremely powerful. You need to trace a workload representing the typical activities in the system against the databases you need to focus on, analyze the trace data, and isolate the processes that need to be tuned.

Before I talk about the specific trace you need to create for such tuning purposes, I'd first like to point out a few important tips regarding working with traces in SQL Server in general.

Traces have an impact on the performance of the system, and you should put effort into reducing their impact. My good friend Brian Moran once compared the problematic aspect of measuring performance to the Heisenberg Uncertainty Principle in quantum mechanics. The principle was formulated by Werner Heisenberg in 1927. Very loosely speaking, when you measure something, there's a factor of uncertainty caused by your measurement. The more precise the measure of something's position, the more uncertainty there is regarding its momentum (loosely, velocity and direction). So the more precisely you know one thing, the less precisely you can know some parallel quantity. On the scale of atoms and elementary particles, the effect of the uncertainty principle is very important. There's no proof to support the uncertainty principal (just like there's no scientific proof of God or of evolution), but the theory is mathematically sound and supported by experimentation.

Going back to our traces, you don't want your tracing activity to cause a performance problem itself. You can't avoid its effect altogether—that's impossible—but you can definitely do much to reduce it by following some important guidelines:

- Don't trace with the SQL Server Profiler GUI; instead, use the T-SQL code that defines the trace. When you trace with Profiler, you're actually running two traces—one that directs the output to the target file, and one that streams the trace information to the client running Profiler. You can define the trace graphically with Profiler, and then script the trace definition to T-SQL code using the menu item File>Export>Script Trace Definition>For SQL Server 2005… (or 2000…).

You can then make slight revisions to the code depending on your needs. I like to encapsulate the code in a stored procedure, which accepts as arguments elements that I want to make variable—for example, the database ID I use as a filter in the trace definition.

- Do not trace directly to a table, as this will have a significant performance impact. Tracing to a file on a local disk is the fastest option (tracing to a network share is bad as well). You can later load the trace data to a table for analysis using the *fn_trace_gettable* function with a SELECT INTO statement. SELECT INTO is considered a bulk operation. Bulk operations are minimally logged when the database recovery model is not set to FULL; therefore, they run much faster then when running in a fully logged mode.
- Tracing can produce enormous amount of data and excessive I/O activity. Make sure the target trace file does not reside on disk drives that contain database files (such as data, log, and tempdb). Ideally, dedicate a separate disk drive for the target trace files, even if it means adding external disk drives.
- Be selective in your choices of event classes and data columns—only trace what you need, removing all default and unnecessary ones. Of course, don't be too selective; make sure that all relevant event classes and data columns are included.
- Use the trace filtering capabilities to filter only the relevant events. For example, when tuning a particular database, make sure you filter events only for the relevant database ID.

With these important guidelines in mind, let's proceed to the trace that we need for our tuning purposes.

Trace Performance Workload

You now need to define a trace that will help you identify processes that need to be tuned in the Performance database. When faced with such a need, there's a tendency to trace slowrunning processes by filtering events where the Duration data column is greater than or equal to some value (say, 3000 milliseconds). This approach is problematic. Think of the following: You have a query that runs for about 30 seconds a couple of times a day, and another query that runs for a about half a second 40,000 times a day. Which would you say is more important to tune? Obviously, the latter is more important, but if you filter only events that run for at least three seconds, you'll filter out the more important query to tune.

In short, for our purposes you don't want to filter based on Duration at all. Of course, this means that you might get enormous amounts of trace data, so make sure you follow the guidelines I suggested earlier. You do want to filter only the databases that are relevant to your tuning process.

As for event classes, if most activities in your system are invoked by stored procedures and each stored procedure invokes a small or limited number of activities, trace the *SP: Completed* event class. You will then be able to aggregate the data by the procedure. However, if each procedure invokes many activities, you want to trace the *SP: StmtCompleted* event class to capture each individual statement invoked from each stored procedure. If you have activities that are submitted as ad hoc batches (as in our case), trace the *SQL: StmtCompleted* event class. Finally, if you have activities submitted as remote procedure calls, trace the *RPC: Completed* event class. Notice that all event classes are *Completed* ones as opposed to the respective *Starting* event classes. Only the *Completed* event classes carry performance information such as Duration, CPU, Reads, and Writes because, naturally, these values are unknown when the respective event starts.

As for data columns, you mainly need the TextData column that will carry the actual T-SQL code, and the relevant performance-related counters—most importantly, the Duration column. Remember that users perceive waits as the performance problem, and Duration stands for the elapsed time it took the event to run. I also like to trace the RowCounts data column, especially when looking for network-related problems. Queries returning the result set to the client with large numbers in this counter would indicate potential pressure on the network. Other than that, you might want additional data columns based on your needs. For example, if you later want to analyze the data by host, application, login, and so on, make sure you also include the corresponding data columns.

You can define a trace following these guidelines, and then script its definition to T-SQL code. I did so, and encapsulated the code in a stored procedure called *sp_perfworkload_trace_start.*

Note Microsoft doesn't recommend using the *sp_* prefix for local user stored procedures. I created the stored procedure with the *sp_* prefix in the master database because it makes the procedure "special" in the sense that you can invoke it from any database without specifying the database qualifier.

The stored procedure accepts a database ID and file name as input parameters. It defines a trace using the specified database ID as a filter, and the given file name as the target for the trace data; it starts the trace, and returns the newly generated trace ID via an output parameter. Run the code in Listing 3-3 to create the *sp_perfworkload_trace_start* stored procedure.

Listing 3-3: Creation script for the *sp_perfworkload_trace_start* stored procedure

```
SET NOCOUNT ON;
USE master;
GO

IF OBJECT_ID('dbo.sp_perfworkload_trace_start') IS NOT NULL
   DROP PROC dbo.sp_perfwork load_trace_start;
GO

CREATE PROC dbo.sp_perfwork load_trace_start
  @dbid      AS INT,
  @tracefile AS NVAR CHAR(254),
  @traceid   AS INT OUTPUT
AS
-- Create a Queue
DECLARE @rc          AS INT;
DECLARE @maxfilesize AS BIGINT;

SET @maxfilesize = 5;

EXEC @rc = sp_trace_create @traceid OUTPUT, 0, @tracefile, @maxfilesize, NULL
IF (@rc != 0) GOTO error;

-- Clientside FileandTable cannot be scripted

-- Set the events
DECLARE @on AS BIT;
SET@on = 1;
EXEC sp_trace_setevent  @traceid,  10,  15, @on;
EXEC sp_trace_setevent  @traceid,  10,  8,  @on;
EXEC sp_trace_setevent  @traceid,  10,  16, @on;
EXEC sp_trace_setevent  @traceid,  10,  48, @on;
EXEC sp_trace_setevent  @traceid,  10,  1,  @on;
EXEC sp_trace_setevent  @traceid,  10,  17, @on;
EXEC sp_trace_setevent  @traceid,  10,  10, @on;
EXEC sp_trace_setevent  @traceid,  10,  18, @on;
EXEC sp_trace_setevent  @traceid,  10,  11, @on;
EXEC sp_trace_setevent  @traceid,  10,  12, @on;
EXEC sp_trace_setevent  @traceid,  10,  13, @on;
EXEC sp_trace_setevent  @traceid,  10,  14, @on;
EXEC sp_trace_setevent  @traceid,  45,  8,  @on;
EXEC sp_trace_setevent  @traceid,  45,  16, @on;
EXEC sp_trace_setevent  @traceid,  45,  48, @on;
EXEC sp_trace_setevent  @traceid,  45,  1,  @on;
EXEC sp_trace_setevent  @traceid,  45,  17, @on;
EXEC sp_trace_setevent  @traceid,  45,  10, @on;
EXEC sp_trace_setevent  @traceid,  45,  18, @on;
EXEC sp_trace_setevent  @traceid,  45,  11, @on;
EXEC sp_trace_setevent  @traceid,  45,  12, @on;
EXEC sp_trace_setevent  @traceid,  45,  13, @on;
EXEC sp_trace_setevent  @traceid,  45,  14, @on;
EXEC sp_trace_setevent  @traceid,  45,  15, @on;
EXEC sp_trace_setevent  @traceid,  41,  15, @on;
EXEC sp_trace_setevent  @traceid,  41,  8,  @on;
EXEC sp_trace_setevent  @traceid,  41,  16, @on;
EXEC sp_trace_setevent  @traceid,  41,  48, @on;
EXEC sp_trace_setevent  @traceid,  41,  1,  @on;
EXEC sp_trace_setevent  @traceid,  41,  17, @on;
EXEC sp_trace_setevent  @traceid,  41,  10, @on;
EXEC sp_trace_setevent  @traceid,  41,  18, @on;
EXEC sp_trace_setevent  @traceid,  41,  11, @on;
EXEC sp_trace_setevent  @traceid,  41,  12, @on;
EXEC sp_trace_setevent  @traceid,  41,  13, @on;
EXEC sp_trace_setevent  @traceid,  41,  14, @on;
```

```
-- Set the Filters
DECLARE @intfilter AS INT;
DECLARE @bigint filter AS BIGINT;
-- Application name filter
EXEC sp_trace_setfilter @traceid, 10, 0, 7, N'SQL Server Profiler%';
-- Database ID filter
EXEC sp_trace_setfilter @traceid, 3, 0, 0, @dbid;

-- Set the trace status to start
EXEC sp_trace_setstatus @traceid, 1;

-- Print trace id and file name for future references
PRINT 'Trce ID:' + CAST(@traceid AS VARCHAR(10))
  + ', Trace File: '''+ @tracefile +'''';

GOTO finish;

error:
PRINT 'ErrorCode: '+ CAST(@rc AS VARCHAR(10));

finish:
GO
```

Run the following code to start the trace, filtering events against the Performance database and sending the trace data to the file ‘c:\temp\Perfworkload 20060828.trc’:

```
DECLARE @dbid AS INT, @traceid AS INT;
SET @dbid = DB_ID('Performance');

EXEC dbo.sp_perfworkload_trace_start
  @dbid      = @dbid,
  @tracefile = 'c:\temp\Perfworkload 20060828.trc',
  @traceid   = @traceid OUTPUT;
```

If you were to assume that the newly generated trace ID is 2, you would get the following output:

```
Trace ID: 2, Trace File: 'c:\temp\perfworkload 20060828.trc'
```

You need to keep the trace ID aside, as you will use it later to stop the trace and close it.

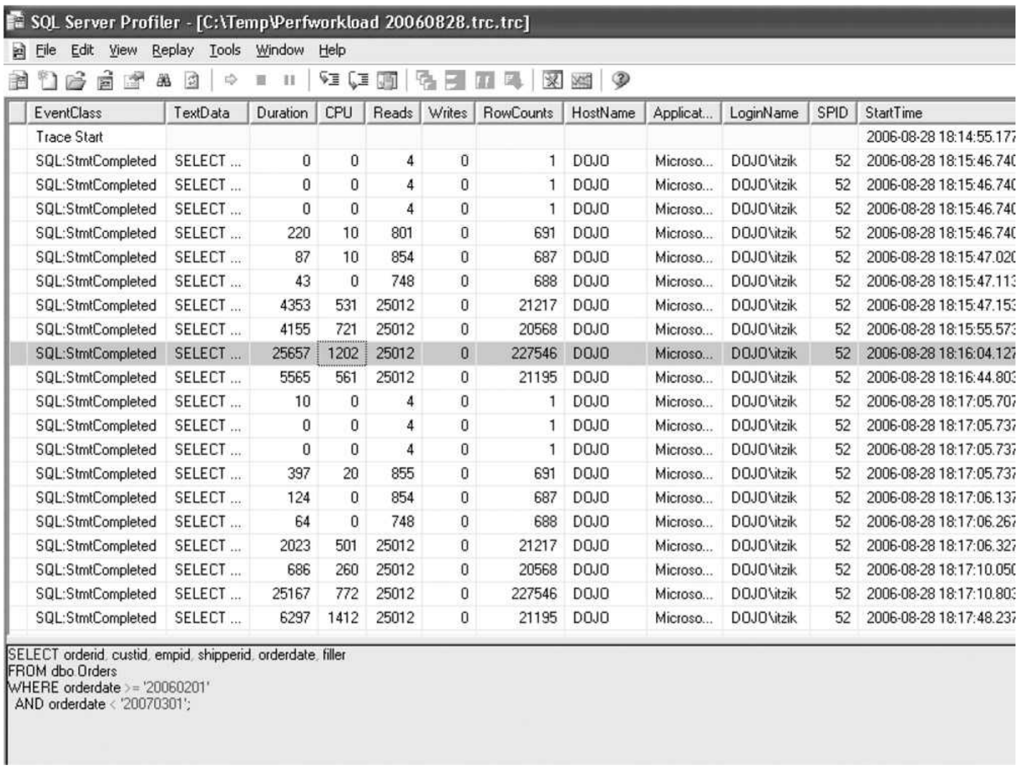
Next, run the sample queries from Listing 3-2 several times. When done, stop the trace and close it by running the following code (assuming the trace ID is 2):

```
EXEC sp_trace_setstatus 2, 0;
EXEC sp_trace_setstatus 2, 2;
```

Of course, you should specify the actual trace ID you got for your trace. If you lost the scrap of paper you wrote the trace ID on, query the sys.traces view to get information about all running traces.

When tracing a workload in a production environment for tuning purposes, make sure you trace a sufficiently representative one. In some cases, this might mean tracing for only a couple of hours, while in other cases it can be a matter of days.

The next step is to load the trace data to a table and analyze it. Of course, you can open it with Profiler and examine it there; however, typically such traces generate a lot of data, and there's not much that you can do with Profiler to analyze the data. In our case, we have a small number of sample queries. Figure 3-4 shows what the trace data looks like when loaded in Profiler.


SQL Server Profiler - [C:\Temp\Perfworkload 20060828.trc.trc]

EventClass	TextData	Duration	CPU	Reads	Writes	RowCounts	HostName	Applicat...	LoginName	SPID	StartTime
Trace Start											2006-08-28 18:14:55.177
SQL:StmtCompleted	SELECT ...	0	0	4	0	1	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:15:46.740
SQL:StmtCompleted	SELECT ...	0	0	4	0	1	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:15:46.740
SQL:StmtCompleted	SELECT ...	0	0	4	0	1	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:15:46.740
SQL:StmtCompleted	SELECT ...	220	10	801	0	691	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:15:46.740
SQL:StmtCompleted	SELECT ...	87	10	854	0	687	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:15:47.020
SQL:StmtCompleted	SELECT ...	43	0	748	0	688	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:15:47.113
SQL:StmtCompleted	SELECT ...	4353	531	25012	0	21217	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:15:47.153
SQL:StmtCompleted	SELECT ...	4155	721	25012	0	20568	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:15:55.573
SQL:StmtCompleted	SELECT ...	25657	1202	25012	0	227546	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:16:04.127
SQL:StmtCompleted	SELECT ...	5565	561	25012	0	21195	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:16:44.803
SQL:StmtCompleted	SELECT ...	10	0	4	0	1	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:05.707
SQL:StmtCompleted	SELECT ...	0	0	4	0	1	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:05.737
SQL:StmtCompleted	SELECT ...	0	0	4	0	1	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:05.737
SQL:StmtCompleted	SELECT ...	397	20	855	0	691	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:05.737
SQL:StmtCompleted	SELECT ...	124	0	854	0	687	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:06.137
SQL:StmtCompleted	SELECT ...	64	0	748	0	688	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:06.267
SQL:StmtCompleted	SELECT ...	2023	501	25012	0	21217	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:06.327
SQL:StmtCompleted	SELECT ...	686	260	25012	0	20568	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:10.050
SQL:StmtCompleted	SELECT ...	25167	772	25012	0	227546	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:10.803
SQL:StmtCompleted	SELECT ...	6297	1412	25012	0	21195	DOJO	Microso...	DOJO\itzik	52	2006-08-28 18:17:48.237

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20070301';
```

Figure 3-4: Performance workload trace data

Examining the trace data, you can clearly see some long-running queries that generate a lot of I/O. These queries use range filters based on the *orderdate* column and seem to consistently incur about 25,000 reads. The Orders table currently contains 1,000,000 rows and resides on about 25,000 pages. This tells you that these queries are causing full table scans to acquire the data and are probably missing an important index on the *orderdate* column. The missing index is probably the main cause of the excessive I/O in the system.

Also, you can find some queries that return a very large number of rows in the result set— several thousand and, in some cases, hundreds of thousands of rows. You should check whether filters and further manipulation are applied in the server when possible, rather than bringing everything to the client through the network and performing filtering and further manipulation there. These queries are probably the main cause of the network issues in the system.

Of course, such graphical analysis with Profiler is feasible only with tiny traces such as the one we're using for demonstration purposes. In production environments, it's just not realistic; you will need to load the trace data to a table and use queries to analyze the data.

Analyze Trace Data

As I mentioned earlier, you use the *fn_trace_gettable* function to return the trace data in table format. Run the following code to load the trace data from our file to the Workload table:

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

SELECT CAST(TextData AS NVARCHAR(MAX)) AS tsql_code,
  Duration AS duration
INTO dbo.Workload
FROM sys.fn_trace_gettable('c:\temp\Perfworkload 20060828.trc', NULL) AS T
WHERE Duration IS NOT NULL;
```

Note that this code loads only the TextData (T-SQL code) and Duration data columns to focus particularly on query run time. Typically, you would want to also load other data columns that are relevant to your analysis—for example, the I/O and CPU counters, row counts, host name, application name, and so on.

Remember that it is important to aggregate the performance information by the query or T-SQL statement to figure out the overall performance impact of each query with its multiple invocations. The following code attempts to do just that, and it generates the output shown in abbreviated form in Table 3-5:

Table 3-5: Aggregated Duration by Query in Abbreviated Form

tsql_code	duration

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '20060118';`
	161055

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '20060212';`
	367430

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '20060828';`
	132466

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate > = '20060101'`
`  AND orderdate < '20060201';`
	3821240

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '20060201'`
`  AND orderdate < '20070301';`
	47881415

...	

```
SELECT
  tsql_code,
  SUM(duration) AS total_duration
FROM dbo.Workload
GROUP BY tsql_code;
```

But there's a problem. You can see in the aggregated data that some queries that are logically the same or follow the same pattern ended up in different groups. That's because they happened to be using different values in their filters. Only query strings that are completely identical were grouped together. As an aside, you wouldn't be facing this problem had you used stored procedures, each invoking an individual query or a very small number of queries. Remember that in such a case you would have traced the *SP: Completed* event class, and then you would have received aggregated data by the procedure. But that's not our case.

A simple but not very accurate way to deal with the problem is to extract a substring of the query strings and aggregate by that substring. Typically, the left portion of query strings that follow the same pattern is the same, while somewhere to the right you have the arguments that are used in the filter. You can apply trial and error, playing with the length of the substring that you will extract; hopefully, the substring will be long enough to allow grouping queries following the same pattern together, and small enough to distinguish queries of different patterns from each other. This approach, as you can see, is tricky and would not guarantee accurate results. Essentially, you pick a number that seems reasonable, close your eyes, and hope for the best.

For example, the following query aggregates the trace data by a query prefix of 100 characters and generates the output shown in Table 3-6:

Table 3-6: Aggregated Duration by Query Prefix

tsql_code	total_duration

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '200`
	660951

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '20`
	60723155

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderid = 3;`
	17857

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 5;
	426

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 7;
	598

SET NO COUNT ON;	7
USE Performance;	12857

```
SELECT
  SUBSTRING(tsql_code,1, 100) AS tsql_code,
  SUM(duration) AS total_duration
FROM dbo.Workload
GROUP BY SUBSTRING(tsql_code,1,100);
```

In our case, this prefix length did the trick for some queries, but it wasn't very successful with others. With more realistic trace data, you won't have the privilege of looking at a tiny number of queries and being able to play with the numbers so easily. But the general idea is that you adjust the prefix length by applying trial and error.

Here's code that uses a prefix length of 94 and generates the output shown in Table 3-7:

Table 3-7: Aggregated Duration by Query Prefix, Adjusted

tsql_code	total_duration

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate
	61384106

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid =
	18881

SET NOCOUNT ON;	7
USE Performance;	12857

```
SELECT
    SUBSTRING(tsql_code, 1,94)AStsql_code,
    SUM(duration) AStotal_duration
FROM dbo.Workload
GROUP BY SUBSTRING(tsql_code,1,94);
```

Now you end up with overgrouping. In short, finding the right prefix length is a tricky process, and its accuracy and reliability is questionable.

A much more accurate approach is to parse the query strings and produce a *query signature* for each. A query signature is a query template that is the same for queries following the same pattern. After creating these, you can then aggregate the data by query signatures instead of by the query strings themselves. SQL Server 2005 provides you with the *sp_get_query_template* stored procedure, which parses an input query string and returns the query template and the definition of the arguments via output parameters.

As an example, the following code invokes the stored procedure, providing a sample query string as input, and it generates the output shown in Table 3-8:

Table 3-8: Query Template

querysig	params
select * from dbo. T1 where col1= @0 and col2 > @1	@0 int,@1 int

```
DECLARE @my_templatetext AS NVARCHAR(MAX);
DECLARE @my_parameters   AS NVARCHAR(MAX);

EXEC sp_get_query_template
  N'SELECT *FROM dbo.T1  WHERE col1 =3 AND col2 > 78',
  @my_templatetext OUTPUT,
  @my_parameters OUTPUT;
```

```
SELECT @my_template text AS querysig, @my_parameters AS params;
```

The problem with this stored procedure is that you need to use a cursor to invoke it against every query string from the trace data, and this can take quite a while with large traces. The stored procedure also (by design) returns an error in some cases (see Books Online for details), which could compromise its value. It would be much more convenient to have this logic implemented as a function, allowing you to invoke it directly against the table containing the trace data. Fortunately, such a function exists; it was written by Stuart Ozer, and its code is provided in Listing 3-4. Stuart is with the Microsoft SQL Server Customer Advisory Team, and I would like to thank him for allowing me to share the code with the readers of this book.

Listing 3-4: Creation script for the *fn_SQLSigT SQL UDF*

```
IF OBJECT_ID('dbo.fn_SQLSigTSQL') IS NOT NULL
  DROP FUNCTION dbo.fn_SQLSigTSQL;
GO

CREATE FUNCTION dbo.fn_SQLSigTSQL
  (@p1 NTEXT, @parselength INT = 4000)
RETURNS NVARCHAR(4000)

--
-- This function is provided "AS IS"  with no warranties,
-- and confers no rights.
-- Use of included script samples are subject to the terms specified at
-- http://www.microsoft.com/info/cpyright.htm
--
-- Strips query strings
AS
BEGIN
  DECLARE @pos AS INT;
  DECLARE @mode AS CHAR(10);
  DECLARE @maxlength AS INT;
  DECLARE @p2 AS NCHAR(4000);
  DECLARE @currchar AS CHAR(1), @nextchar AS CHAR(1);
  DECLARE @p2len AS INT;

  SET @maxlength = LEN(RTRIM(SUBSTRING(@p1,1,4000)));
  SET @maxlength = CASE WHEN  @maxlength > @parselength
                   THEN  @parselength ELSE  @maxlength END;
  SET @pos = 1;
  SET @p2 = '';
  SET @p2len = 0;
  SET @currchar = '';
  set @nextchar = '';
  SET @mode = 'command';

  WHILE  (@pos <= @maxlength)
    BEGIN
      SET @currchar = SUBSTRING(@p1,@pos,1);
      SET @nextchar = SUBSTRING(@p1,@pos+1,1);
      IF @mode = 'command'
      BEGIN
        SET @p2 = LEFT(@p2,@p2len) + @currchar;
        SET @p2len = @p2len  + 1;
        IF @currchar IN (',','(','  ','=','<','>','!')
          AND @nextchar BETWEEN '0' AND '9'
        BEGIN
          SET @mode = 'number';
          SET @p2 = LEFT(@p2,@p2len) + '#';
          SET @p2len = @p2len + 1;
        END
        IF@currchar = ''''
        BEGIN
          SET @mode = 'literal';
          SET @p2 = LEFT(@p2,@p2len) + '#''';
          SET @p2len = @p2len + 2;
        END
      END
      ELSE IF @mode = 'number' AND @nextchar IN(',',')',' ','=','<','>','!')
       SET@mode='command';
      ELSE IF @mode = 'literal' AND @currchar = ''''
       SET @mode = 'command';

      SET @pos = @pos +  1;
    END
    RETURN @p2;
  END
  GO
```

The function accepts as inputs a query string and the length of the code you want to parse. The function returns the query signature of the input query, with all parameters replaced by anumber sign (#). Note that this is a fairly simple function and might need to be tailored to particular situations. Run the following code to test the function:

```
SELECT dbo.fn_SQLSigTSQL
  (N'SELECT * FROM dbo.T1 WHERE col1 = 3 AND col2 > 78', 4000);
```

You will get the following output:

```
SELECT * FROM dbo.T1 WHERE col1 = # AND col2 > #
```

Of course, you could now use the function and aggregate the trace data by query signature. However, keep in mind that although T-SQL is very efficient with data manipulation, it is slow in processing iterative/procedural logic. This is a classic example where a CLR implementation of the function makes more sense. The CLR is much faster than T-SQL for iterative/procedural logic and string manipulation. SQL Server 2005 introduces.NET integration within the product, allowing you to develop.NET routines based on the *common language runtime* (CLR). CLR routines are discussed in *Inside T-SQL Programming*; there you will find more thorough coverage and a comparison of the T-SQL and CLR-based implementations of the function that generates query signatures. You can also find some background information about CLR development in SQL Server in Chapter 6 of this book.

The CLR-based "enhanced" implementation of the function using C# code is provided in Listing 3-5.

Listing 3-5: *fn_SQLSigCLR* and *fn_RegexReplace* functions, C# version

```
using System.Text;
using Microsoft.SqlServer.Server;
using System.Data.SqlTypes;
using System.Text.Regular Expressions;

public partial class SQLSignature
{
    //fn_SQLSigCLR
    [SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
    public static SqlString fn_SQLSigCLR(SqlString querystring)
    {
        return (SqlString)Regex.Replace(
            querystring.Value,
            @"([\s,(=<>!](?![^\]]+[\]]))(?:(?:(?:(?#  expression coming
             )(?:([N])?(')(?:[^']|'')*('))(?#                       character
             )|(?:0x[\da-fA-F]*)(?#                                 binary
             )|(?:[-+]?(?:(?:[\d]*\.[\d]*|[\d]+)(?#                 precise number
             )(?:[eE]?[\d]*)))(?#                                   imprecise number
             )|(?:[~]?[-+]?(?:[\d]+))(?#                            integer
             ))(?:[\s]?[\+\-\*\/\%\&\|\^][\s]?)?)+(?#        operators
             ))",
            @"$1$2$3#$4");
     }
    // fn_RegexReplace - for generic useof RegEx-based replace
    [SqlFunction(IsDeterministic = true, DataAccess = DataAccessKind.None)]
    public static SqlString fn_RegexReplace(
        SqlString input, SqlString pattern, SqlString replacement)
    {
        return  (SqlString)Regex.Replace(
            input.Value, pattern.Value, replacement.Value);
    }
```

```
}
```

The code in Listing 3-5 has the definitions of two functions: *fn_SQLSigCLR* and *fn_RegexReplace.* The function *fn_SQLSigCLR* accepts a query string and returns the query signature. This function covers cases that the T-SQL function overlooks, and it can be easily enhanced to support more cases if you need it to. The function *fn_RegexReplace* exposes more generic pattern-based string replacement capabilities based on regular expressions, and it is provided for your convenience.

> **Note** I didn't bother checking for NULL inputs in the CLR code because T-SQL allows you to specify the option RETURNS NULL ON NULL INPUT when you register the functions, as I will demonstrate later. This option means that when a NULL input is provided, SQL Server doesn't invoke the function at all; rather, it simply returns a NULL output.

If you're familiar with developing CLR routines in SQL Server, deploy these functions in the Performance database. If you're not, just follow these steps:

1. Create a new Microsoft Visual C#, Class Library project in Microsoft Visual Studio 2005 (File>New>Project…>Visual C#>Class Library).

2. In the New Project dialog box, name the project and solution **SQLSignature**, specify C:\ as the location, and confirm.

3. Rename the file Class1.cs to **SQLSignature.cs**, and within it paste the code from Listing 3-5, overriding its current content.

4. Build the assembly by choosing the Build>Build SQLSignature menu item. A file named C:\SQLSignature\SQLSignature\bin\Debug\SQLSignature.dll containing the assembly will be created.

5. At this point, you go back to SQL Server Management Studio (SSMS) and apply a couple of additional steps to deploy the assembly in the Performance database, and then register the *fn_SQLSigCLR* and *fn_RegexReplace* functions. But first, you need to enable CLR in SQL Server (which is disabled by default) by running the following code:

   ```
   EXEC sp_configure 'clr enable', 1;
   RECONFIGURE;
   ```

6. Next, you need to load the intermediate language (IL) code from the .dll file into the Performance database by running the following code:

   ```
   USE Performance;
   CREATE ASSEMBLY SQLSignature
   FROM 'C:\SQLSignature\SQLSignature\bin\Debug\SQLSignature.dll';
   ```

7. Finally, register the *fn_SQLSigCLR* and *fn_RegexReplace* functions by running the following code:

   ```
   CREATE FUNCTION dbo.fn_SQLSigCLR(@querystring AS NVARCHAR(MAX))
   RETURNS NVARCHAR(MAX)
   WITH RETURNS NULL ON NULL INPUT
   EXTERNAL NAME SQLSignature. SQLSignature.fn_SQLSigCLR;
   GO

   CREATE FUNCTION dbo.fn_RegexReplace(
     @input       AS   NVARCHAR(MAX),
     @pattern     AS   NVARCHAR(MAX),
     @replacement AS   NVARCHAR(MAX))
   RETURNS NVARCHAR(MAX)
   WITH RETURNS NULL ON NULL INPUT
   EXTERNAL NAME SQLSignature. SQLSignature.fn_RegexReplace;
   GO
   ```

You're done. At this point, you can start using the functions like you do any other user-defined function. In case you're curious, the CLR implementation of the function runs faster than the T-SQL one by a factor of 10.

To test the *fn_SQLSigCLR* function, invoke it against the Workload table by running the following query, which will generate the output shown in abbreviated form in Table 3-9:

Table 3-9: Trace Data with Query Signatures in Abbreviated Form

sig	duration
…	

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderid = #;`
	17567

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderid = #;`
	72

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderid = #;`
	145

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '#';`
	99204

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '#';`
	56191

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate = '#';`
	38718

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '#'`
`  AND orderdate < '#';`
	1689740

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '#'`
`  AND orderdate < '#';`
	738292

`SELECT orderid, custid, empid, shipperid, orderdate, filler`
`FROM dbo.Orders`
`WHERE orderdate >= '#'`
`  AND orderdate < '#';`
	13601583

```
SELECT
  dbo.fn_SQLSigCLR(tsql_code) AS sig,
  duration
FROM dbo.Workload;
```

You can also use the more generic *fn_Regex Replace* function to achive the same output, like so:

```
SELECT
  dbo.fn_Regex Replace(tsql_code,
    N'([\s,(=<>!](?![^\]]+[\]]))(?:(?:(?:(?#     expression coming
     )(?:([N])?('')(?:[^'']|'''')*(''))(?#                     character
     )|(?:0x[\da-fA-F]*)(?#                                    binary
     )|(?:[-+]?(?:(?:[\d]*\.[\d]*|[\d]+)(?#                    precise number
     )(?:[eE]?[\d]*)))(?#                                      imprecise number
     )|(?:[~]?[-+]?(?:[\d]+))(?#                               integer
     ))(?:[\s]?[\+\-\*\/\%\&\|\^][\s]?)?)+(?#           operators
     ))',
    N'$1$2$3#$4') AS sig,
  duration
FROM dbo.Workload;
```

As you can see, you get back query signatures, which you can use to aggregate the trace data. Keep in mind, though, that query strings can get lengthy, and grouping the data by lengthy strings is slow and expensive. Instead, you might prefer to generate an integer checksum for each query string by using the T-SQL CHECKSUM function. For example, the following query generates a checksum value for each query string from the Workload table, and it generates the output shown in abbreviated form in Table 3-10:

Table 3-10: Query

Signature Checksums in Abbreviated Form

cs	duration
…	
–1872968693	17567
–1872968693	72
–1872968693	145
–184235228	99204
–184235228	56191
–184235228	38718
368623506	1689740
368623506	738292
368623506	13601583
…	

```
SELECT
  CHECKSUM(dbo.fn_SQLSigCLR(tsql_code)) AS cs,
  duration
FROM dbo.Workload;
```

Use the following code to add a column called *cs* to the Workload table, populate it with the checksum of the query signatures, and create a clustered index on the *cs* column:

```
ALTER TABLE dbo.Workload ADD cs INT NOT NULL DEFAULT (0);
GO
UPDATE dbo.Workload
  SET cs = CHECKSUM(dbo.fn_SQLSigCLR(tsql_code));

CREATE CLUSTERED INDEX idx_cl_cs ON dbo.Workload(cs);
```

Run the following code to return the new contents of the Workload table, shown in abbreviated form in Table 3-11:

Table 3-11: Contents of Table Workload

tsql_code	duration	cs
`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate = '20060118';`	36094	–184235228
`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate = '20060828';`	32662	–184235228
`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060101' AND orderdate < '20060201';`	717757	368623506
`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060401' AND orderdate < '20060501';`	684754	368623506
…		

```
SELECT tsql_code, duration, cs
FROM dbo.Workload
```

At this point, you want to aggregate the data by the query signature checksum. It would also be very useful to get running aggregates of the percentage of each signature's duration of the total duration. This information can help you easily isolate the query patterns that you need to tune. Remember that typical production workloads can contain a large number of query signatures. It would make sense to populate a temporary table with the aggregate data and index it, and then run a query against the temporary table to calculate the running aggregates.

Run the following code to populate the temporary table #AggQueries with the total duration per signature checksum, including the percentage of the total, and a row number based on the duration in descending order:

```
IF OBJECT_ID('tempdb..#AggQueries') IS NOT NULL
  DROP TABLE  #AggQueries;
GO

SELECT cs, SUM(duration) AS total_duration,
  100. * SUM(duration) / SUM(SUM(duration))OVER() AS pct,
  ROW_NUMBER() OVER(ORDER BY SUM(duration) DESC)  AS rn
INTO #AggQueries
FROM dbo.Workload
GROUP BY cs;

CREATE CLUSTERED INDEX idx_cl_cs ON #AggQueries(cs);
```

Run the following code, which generates the output shown in Table 3-12, to return the contents of the temporary table:

Table 3-12: Aggregated Duration by Query Signature Checksum

cs	total_duration	pct	rn
368623506	60723155	98.872121791489952	1
−184235228	660951	1.076189598024132	2
−1872968693	18881	0.030742877762941	3
1018047893	12857	0.020934335013936	4
1037912028	7	0.000011397709037	5

```
SELECT cs, total_duration, pct, rn
FROM #AggQueries
ORDER BY rn;
```

Use the following query to return the running aggregates of the percentages, filtering only those rows where the running percentage accumulates to a certain threshold that you specify:

```
SELECT AQ1.cs,
  CAST(AQ1.total_duration / 1000.
    AS DECIMAL(12, 2))AS total_s,
  CAST(SUM(AQ2.total_duration) / 1000.
    AS DECIMAL(12, 2)) AS running_total_s,
  CAST(AQ1.pct AS DECIMAL(12,2))AS pct,
  CAST(SUM(AQ2.pct)AS DECIMAL(12, 2)) AS run_pct,
  AQ1.rn
FROM #AggQueries AS AQ1
  JOIN  #AggQueriesASAQ2
    ON AQ2.rn < = AQ1.rn
GROUP BY AQ1.cs, AQ1.total_duration, AQ1.pct, AQ1.rn
HAVING SUM(AQ2.pct) - AQ1.pct<= 90 -- percentage threshold
--  OR AQ1.rn < = 5
ORDER BY AQ1.rn;
```

In our case, if you use 90 percent as the threshold, you would get only one row. For demonstration purposes, I uncommented the part of the expression in the HAVING clause to return at least 5 rows and got the output shown in Table 3-13.

Table 3-13: Running Aggregated Duration

cs	total_s	running_total_s	pct	running_pct	rn
368623506	60723.16	60723.16	98.87	98.87	1
−184235228	660.95	61384.11	1.08	99.95	2
−1872968693	18.88	61402.99	0.03	99.98	3
1018047893	12.86	61415.84	0.02	100.00	4

1037912028	0.01	61415.85	0.00	100.00	5

You can see at the top that there's one query pattern that accounts for 98.87 percent of the total duration. Based on my experience, typically a handful of query patterns cause most of the performance problems in a given system.

To get back the actual queries that you need to tune, you should join the result table returned from the preceding query with the Workload table, based on a match in the checksum value (*cs* column), like so:

```
WITH RunningTotals AS
(
  SELECT AQ1.cs,
    CAST(AQ1.total_duration / 1000.
      AS DECIMAL(12, 2))AS total_s,
    CAST(SUM(AQ2.total_duration) /  1000.
      AS DECIMAL(12, 2)) AS running_total_s,
    CAST(AQ1.pct AS DECIMAL(12,2))AS pct,
    CAST(SUM(AQ2.pct) AS DECIMAL(12,2)) AS run_pct,
    AQ1.rn
  FROM #AggQueries AS AQ1
    JOIN  #AggQueries AS AQ2
      ON AQ2.rn <= AQ1.rn
  GROUP BY AQ1.cs, AQ1.total_duration, AQ1.pct, AQ1.rn
  HAVING SUM(AQ2.pct) - AQ1.pct< = 90 -- percentage threshold
  --  OR AQ1.rn <= 5
)
 SELECT RT.rn, RT.pct, W.tsql_code
 FROM Running Totals AS RT
   JOIN dbo.Workload AS W
     ON W.cs = RT.cs
 ORDER BY RT.rn;
```

You will get the output shown in abbreviated form in Table 3-14.

Table 3-14: Top Slow Queries in Abbreviated Form

rn	pct	tsql_code
1	98.87	`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060101' AND orderdate < '20060201';`
1	98.87	`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060401' AND orderdate < '20060501';`
1	98.87	`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060201' AND orderdate < '20070301';`
…		

Of course, with a more realistic workload you might get a large number of queries back, but you're really interested in the query pattern that you need to tune. So instead of joining back to the Workload table, use the APPLY operator to return only one row for each query signature with the query pattern, and a single sample per pattern out of the actual queries like so:

```
WITH RunningTotals AS
(
  SELECT AQ1.cs,
    CAST(AQ1.total_duration / 1000.
      AS DECIMAL(12, 2)) AS total_s,
    CAST(SUM(AQ2.total_duration) / 1000.
      AS DECIMAL (12, 2)) AS running_total_s,
    CAST(AQ1.pct AS DECIMAL(12, 2)) AS pct,
    CAST(SUM(AQ2.pct) AS DECIMAL(12, 2)) AS run_pct,
        AQ1.rn
  FROM #AggQueries AS AQ1
    JOIN#AggQueries AS AQ2
      ONAQ2.rn <= AQ1.rn
  GROUPBY AQ1.cs, AQ1.total_duration, AQ1.pct, AQ1.rn
  HAVING SUM(AQ2.pct)-  AQ1.pct <= 90 -- percentage threshold
)
SELECT RT.rn, RT.pct, S.sig, S.tsql_code AS sample_query
FROM RunningTotals AS RT
  CROSS APPLY
    (SELECT TOP(1) tsql_code, dbo.fn_SQLSigCLR (tsql_code) ASsig
       FROM dbo.Workload AS W
       WHERE W.cs = RT.cs) AS S
ORDER BY RT.rn;
```

You will get the output shown in Table 3-15.

Table 3-15: Signature and Sample of the Top Slow Queries

rn	pct	sig	sample_query
1	98.87	`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '#' AND orderdate < '#';`	`SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WHERE orderdate >= '20060101' AND orderdate < '20060201';`

Now you can focus your tuning efforts on the query patterns that you got back—in our case, only one. Of course, in a similar manner you can identify the query patterns that generate the largest result sets, most of the I/O, and so on.

Tune Indexes/Queries

Now that you know which patterns you need to tune, you can start with a more focused query-tuning process. The process might involve index tuning or query code revisions, and we will practice it thoroughly throughout the book. Or you might realize that the queries are already tuned pretty well, in which case you would need to inspect other aspects of the system (for example, hardware, database layout, and so on).

In our case, the tuning process is fairly simple. You need to create a clustered index on the *orderdate* column:

```
CREATE CLUSTERED INDEX idx_cl_od ON dbo.Orders(orderdate);
```

Later in the chapter, I'll cover index tuning and explain why a clustered index is adequate for query patterns such as the ones that our tuning process isolated.

To see the effect of adding the index, run the following code to start a new trace:

```
DECLARE @dbid AS INT, @traceid AS INT;
SET @dbid = DB_ID('Performance');

EXEC dbo.sp_perfworkload_trace_start
  @dbid      = @dbid,
  @tracefile = 'c:\temp\Perfworkload 20060829',
  @traceid   = @traceid OUTPUT;
```

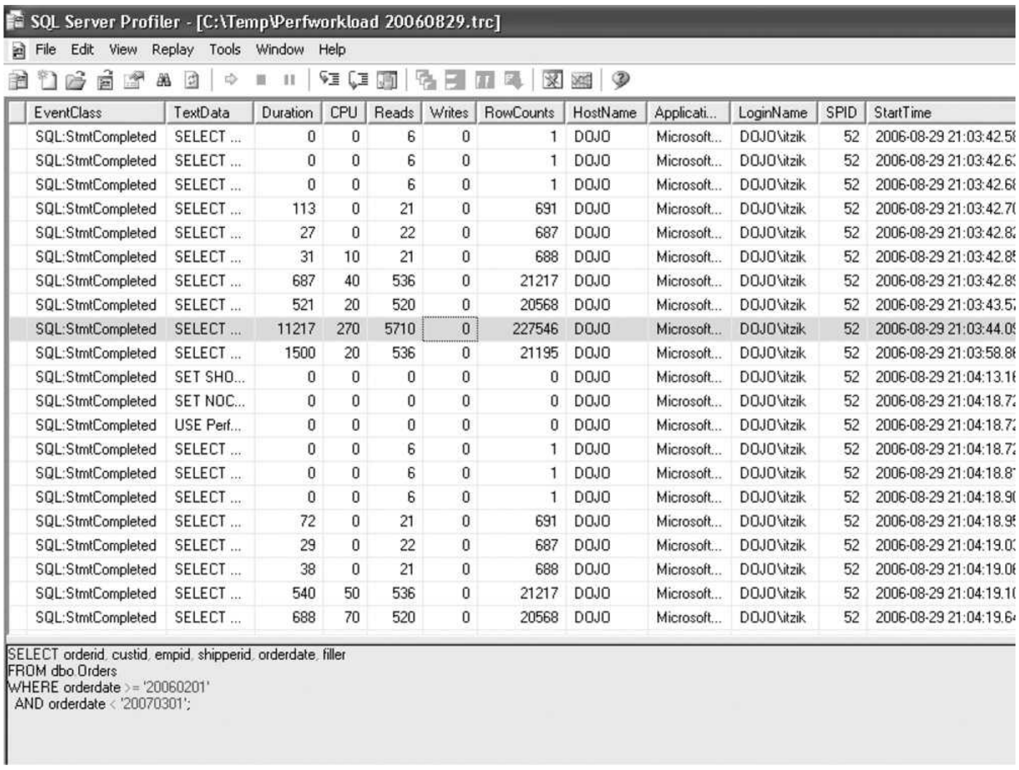
When I ran this code, I got the following output showing that the trace ID generated is 2:

```
Trace ID: 2, Trace File: 'c:\temp\Perfworkload 20060829.trc'
```

Run the sample queries from Listing 3-2 again, and then stop the trace:

```
EXEC sp_trace_setstatus 2, 0;
EXEC sp_trace_setstatus 2, 2;
```

Figure 3-5 shows the trace data loaded with Profiler.


SQL Server Profiler - [C:\Temp\Perfworkload 20060829.trc]
File Edit View Replay Tools Window Help

EventClass	TextData	Duration	CPU	Reads	Writes	RowCounts	HostName	Applicati...	LoginName	SPID	StartTime
SQL:StmtCompleted	SELECT ...	0	0	6	0	1	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:42.5
SQL:StmtCompleted	SELECT ...	0	0	6	0	1	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:42.6
SQL:StmtCompleted	SELECT ...	0	0	6	0	1	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:42.6
SQL:StmtCompleted	SELECT ...	113	0	21	0	691	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:42.7
SQL:StmtCompleted	SELECT ...	27	0	22	0	687	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:42.8
SQL:StmtCompleted	SELECT ...	31	10	21	0	688	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:42.8
SQL:StmtCompleted	SELECT ...	687	40	536	0	21217	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:42.8
SQL:StmtCompleted	SELECT ...	521	20	520	0	20568	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:43.5
SQL:StmtCompleted	SELECT ...	11217	270	5710	0	227546	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:44.0
SQL:StmtCompleted	SELECT ...	1500	20	536	0	21195	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:03:58.8
SQL:StmtCompleted	SET SHO...	0	0	0	0	0	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:13.1
SQL:StmtCompleted	SET NOC...	0	0	0	0	0	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:18.7
SQL:StmtCompleted	USE Perf...	0	0	0	0	0	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:18.7
SQL:StmtCompleted	SELECT ...	0	0	6	0	1	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:18.7
SQL:StmtCompleted	SELECT ...	0	0	6	0	1	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:18.8
SQL:StmtCompleted	SELECT ...	0	0	6	0	1	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:18.9
SQL:StmtCompleted	SELECT ...	72	0	21	0	691	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:18.9
SQL:StmtCompleted	SELECT ...	29	0	22	0	687	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:19.0
SQL:StmtCompleted	SELECT ...	38	0	21	0	688	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:19.0
SQL:StmtCompleted	SELECT ...	540	50	536	0	21217	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:19.1
SQL:StmtCompleted	SELECT ...	688	70	520	0	20568	DOJO	Microsoft...	DOJO\itzik	52	2006-08-29 21:04:19.6

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060201'
 AND orderdate < '20070301';
```

Figure 3-5: Performance workload trace data after adding index

You can see that the duration and I/O involved with the query pattern we tuned are greatly reduced. Still, there are queries that generate a lot of network traffic. With those, you might want to check whether some of the processing of their result sets could be achieved at the server side, thus reducing the amount of data submitted through the network.

Tools for Query Tuning

This section provides an overview of the query-tuning tools that will be used throughout these books, and it will focus on analyzing execution plans.

syscacheobjects

SQL Server 2000 provides you with a virtual system table called master.dbo.syscacheobjects, which contains information about cached execution plans. SQL Server 2005 provides you with a compatibility view called *sys.syscacheobjects*, and a new DMV and two DMFs that replace the legacy system table and compatibility view. The three new objects are: *sys.dm_exec_cached_plans*, *sys.dm_exec_plan_attributes*, and *sys.dm_exec_sql_text.* These give you extremely useful information when analyzing the caching, compilation, and recompilation behavior of execution plans. The *sys.dm_exec_cached_plans* object contains information about the cached query execution plans; *sys.dm_exec_plan_attributes* contains one row per attribute associated with the plan, whose handle is provided as input to the DMF; *sys.dm_exec_sql_text* returns the text associated with the query, whose handle is provided as input to the DMF.

Clearing the Cache

When analyzing query performance, you sometimes need to clear the cache. SQL Server provides you with tools to clear both data and execution plans from cache. To clear data from cache globally, use the following command:

```
DBCC DROPCLEANBUFFERS;
```

To clear execution plans from cache globally, use the following command:

```
DBCC FREEPROCCACHE;
```

To clear execution plans of a particular database, use the following command:

```
DBCC FLUSHPROCINDB(<db_id>);
```

Note that the DBCC FLUSHPROCINDB command is undocumented.

Caution Be careful not to use these commands in production environments because, obviously, clearing the cache will have a performance impact on the whole system. After clearing the data cache, SQL Server will need to physically read pages accessed for the first time from disk. After clearing execution plans from cache, SQL Server will need to generate new execution plans for queries. Also, be sure that you are aware of the global impact of clearing the cache even when doing so in development or test environments.

Dynamic Management Objects

SQL Server 2005 introduces more than 70 dynamic management objects, including DMVs and DMFs. These contain extremely useful information about the server that you can use to monitor SQL Server, diagnose problems, and tune performance. Much of the information provided by these views and functions was never available in the past. It is time very well spent to study them in detail. In these books, I will make use of the ones that are relevant to my discussions, but I urge you to take a close look at others as well. You can find information about them in Books Online, and I will also give you pointers to additional information at the end of the chapter.

STATISTICS IO

STATISTICS IO is a session option that is used extensively throughout these books. It returns I/O-related information about the statements that you run. To demonstrate its use, first clear the data cache:

```
DBCC DROPCLEANBUFFERS;
```

Then run the following code to turn the session option on and invoke a query:

```
SET STATISTICS IO ON;

SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060101'
  AND orderdate < '20060201';
```

You should get output similar to the following:

```
Table  'Orders'.  Scan count  1, logical reads 536, physical reads 2, read-
ahead reads 532, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

The output tells you how many times the table was accessed in the plan (*Scan count*); how many reads from cache were involved (*logical reads*); how many reads from disk were involved (*physical reads* and *read-ahead reads*); and similarly, how many logical and physical reads related to large objects were involved (*lob logical reads*, *lob physical reads*, *lob readahead reads*).

Run the following code to turn the session option off:

```
SET STATISTICS IO OFF;
```

Measuring the Run Time of Queries

STATISTICS TIME is a session option that returns the net CPU and elapsed clock time information about the statements that you run. It returns this information for both the time it took to parse and compile the query, and the time it took to execute it. To demonstrate the use of this session option, first clear both the data and execution plans from cache:

```
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
```

Run the following code to turn the session option on:

```
SET STATISTICS TIME ON;
```

Then invoke the following query:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderdate >= '20060101'
  AND orderdate > '20060201';
```

You will get output similar to the following:

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 30 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.

SQL Server Execution Times:
   CPU time = 30 ms, elapsed time = 619 ms.
```

The output tells you the net CPU time and elapsed clock time for parsing and compiling the query, and also the time it took to execute it. Run the following code to turn the option off:

```
SET STATISTICS TIME OFF;
```

This tool is convenient when you want to analyze the performance of an individual query interactively. When you run benchmarks in batch mode, the way to measure the run time of queries is different. Store the value of the GETDATE function in a variable right before the query. Right after the query, issue an INSERT statement into the table where you collect performance information, subtracting the value stored in the variable from the current value of GETDATE. Note that GETDATE returns a DATETIME value, which has an accuracy level of 3.33 milliseconds. When measuring the time statistics of queries for which this accuracy level is insufficient, run the queries repeatedly in a loop and divide run time for the entire loop by the number of iterations.

Analyzing Execution Plans

An execution plan is the "work plan" the optimizer generates to determine how to process a given query. The plan contains operators that are generally applied in a specific order. Some operators can be applied while their preceding operator is still in progress. Some operators might be applied more than once. Also, some branches of the plan are invoked in parallel if the optimizer chose a parallel plan. In the plan, the optimizer determines the order in which to access the tables involved in the query, which indexes to use and which access methods to use to apply to them, which join algorithms to use, and so on. In fact, for a given query the optimizer considers multiple execution plans, and it chooses the plan with the lowest cost out of the ones that were generated. Note that SQL Server might not generate all possible execution plans for a given query. If it always did, the optimization process could take too long. SQL Server will calculate a cost threshold for the optimization process based on the sizes of the tables involved in the query, among other things. At the end of the chapter, I'll point you to a white paper that provides detailed information about this process.

Throughout these books, I'll frequently analyze execution plans of queries. This section and the one that follows ("Index Tuning") should give you the background required to follow and understand the discussions involving plan analysis. Note that the purpose of this section is not to get you familiarized with all possible operators, rather it is to familiarize you with the techniques to analyze plans. The "Index Tuning" section will familiarize you with indexrelated operators, and later in the book I'll elaborate on additional operators—for example, join-related operators will be described in Chapter 5.

Graphical Execution Plans

Graphical execution plans are used extensively throughout these books. SSMS allows you to get both an estimated execution plan (by pressing Ctrl+L) and to include an actual one (by pressing Ctrl+M) along with the output of the query you run. Note that both will give you the same plan; remember that an execution plan is generated before the query is run. However, when you request an estimated plan, the query is not run at all. Obviously, some measures can be collected only at run time (for example, the actual number of rows returned from each operator, and the actual number of rebinds and rewinds). In the estimated plan, you will see estimations for measures that can be collected only at run time, while the actual plan will show the actuals and also some of the same estimates.

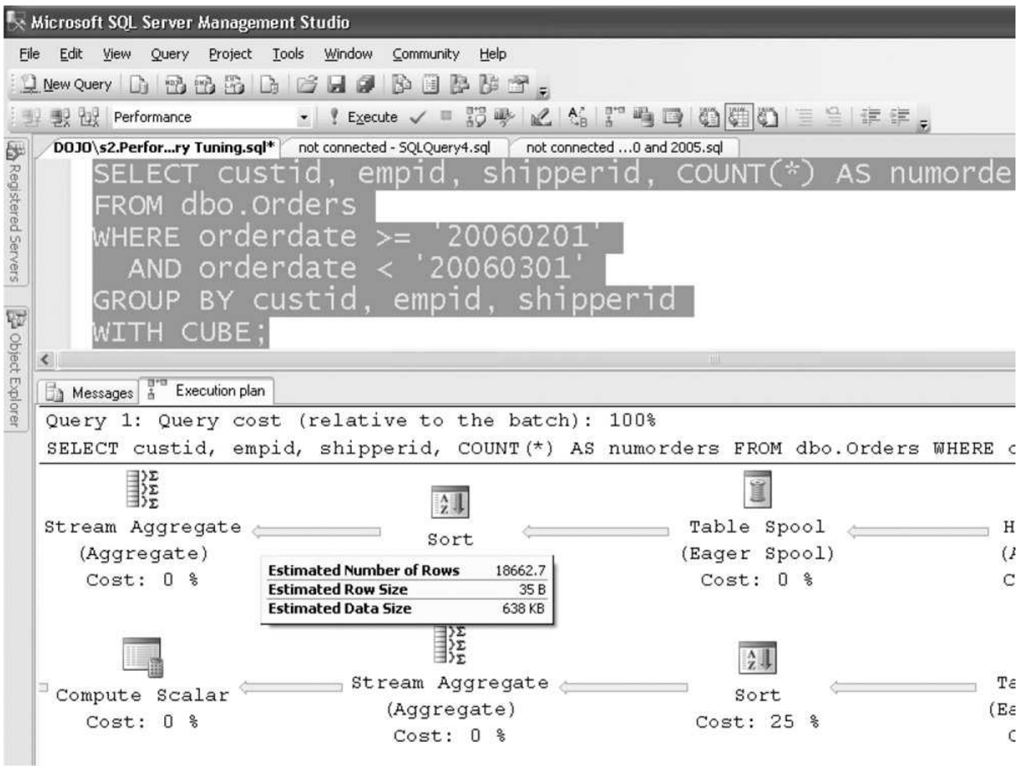
To demonstrate a graphical execution plan analysis, I will use the following query:

```
SELECT custid, empid, shipperid, COUNT(*) AS numorders
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH CUBE;
```

The query returns aggregated counts of orders for a given period, grouped by *custid*, *empid*, and *shipperid*, including the calculation of super aggregates (generated by the CUBE option). I'll discuss CUBE queries in detail in Chapter 6.

Note I did some graphical manipulation on the execution plans that appear in this chapter to fit images in the printed pages and for clarity.

As an example, if you request an estimated execution plan for the preceding query, you will get the plan shown in Figure 3-6.



```
SELECT custid, empid, shipperid, COUNT(*) AS numorde
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH CUBE;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT custid, empid, shipperid, COUNT(*) AS numorders FROM dbo.Orders WHERE c
```

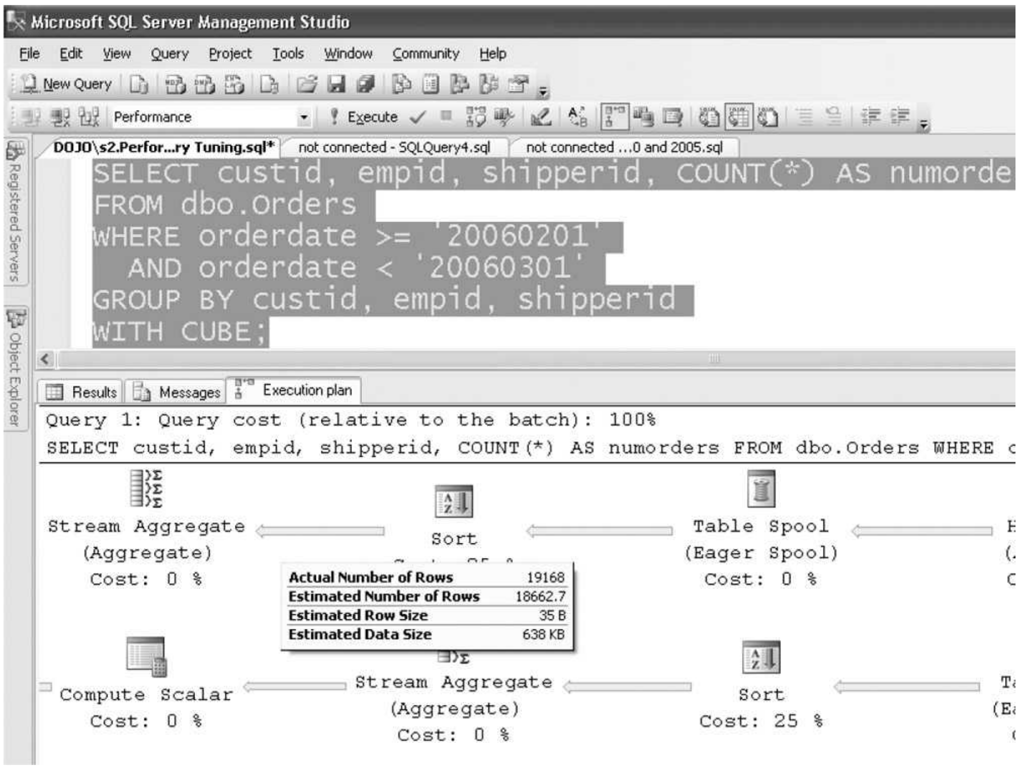
Estimated Number of Rows	18662.7
Estimated Row Size	35 B
Estimated Data Size	638 KB

Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Community Help
New Query
Performance
Execute
DOJO\s2.Perfor...ry Tuning.sql*
not connected - SQLQuery4.sql
not connected ...0 and 2005.sql
Registered Servers
Object Explorer
Messages
Execution plan
Stream Aggregate (Aggregate) Cost: 0 %
Sort
Table Spool (Eager Spool) Cost: 0 %
Compute Scalar Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %
Sort Cost: 25 %

Figure 3-6: Estimated execution plan example

Notice that when you place your mouse pointer over an arrow that goes out of an operator (for example, the one going out of the first Sort operator), you get an estimated number of rows. By the way, a nice aspect of the arrows representing data flow is that their thickness is proportional to the number of rows returned by the source operator. You want to keep an eye especially on thick arrows, as these might indicate a performance issue.

Next, turn on the Include Actual Execution Plan option, and run the query. You will get both the output of the query and the actual plan, as shown in Figure 3-7.



```
SELECT custid, empid, shipperid, COUNT(*) AS numorde
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH CUBE;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT custid, empid, shipperid, COUNT(*) AS numorders FROM dbo.Orders WHERE c
```

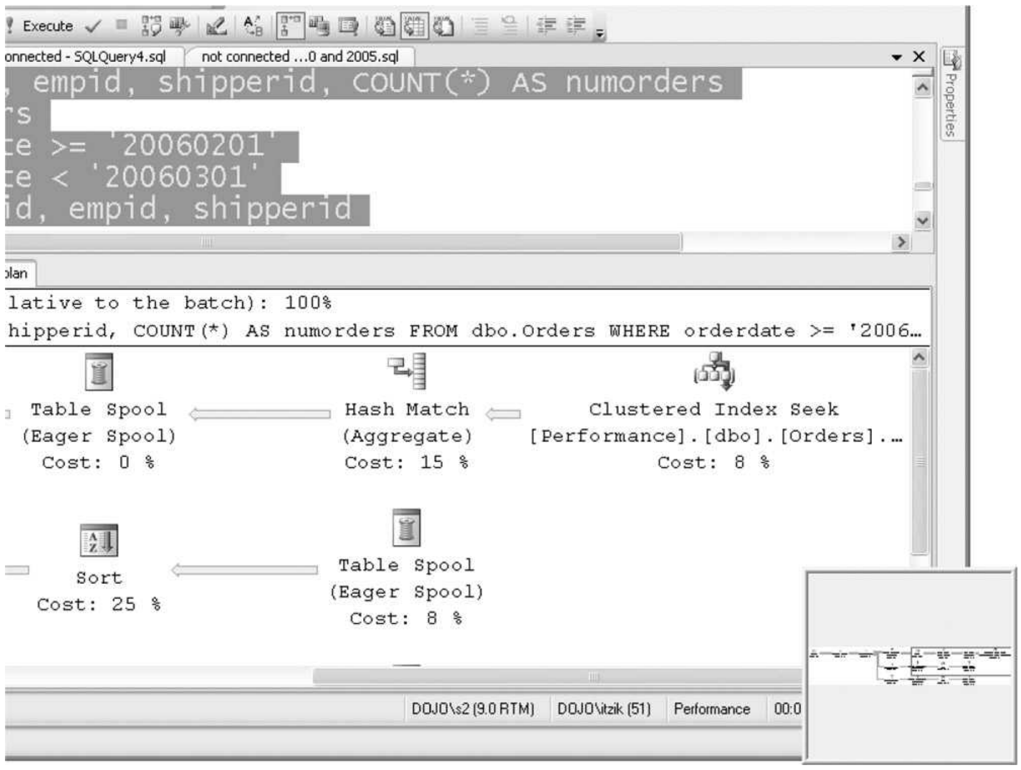
Actual Number of Rows	19168
Estimated Number of Rows	18662.7
Estimated Row Size	35 B
Estimated Data Size	638 KB

Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Community Help
New Query
Performance
Execute
DOJO\s2.Perfor...ry Tuning.sql*
not connected - SQLQuery4.sql
not connected ...0 and 2005.sql
Registered Servers
Object Explorer
Results
Messages
Execution plan
Stream Aggregate (Aggregate) Cost: 0 %
Sort
Table Spool (Eager Spool) Cost: 0 %
Compute Scalar Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %
Sort Cost: 25 %

Figure 3-7: Actual execution plan example

Notice that now you get the actual number of rows returned by the source operator.

When you get elaborated plans like this one that do not fit in one screen, you can use a really cool new zooming feature. Press the + button that appears at the bottom right corner of the execution plan pane, and you will get a rectangle that allows you to navigate to a desired place in the plan, as shown in Figure 3-8.

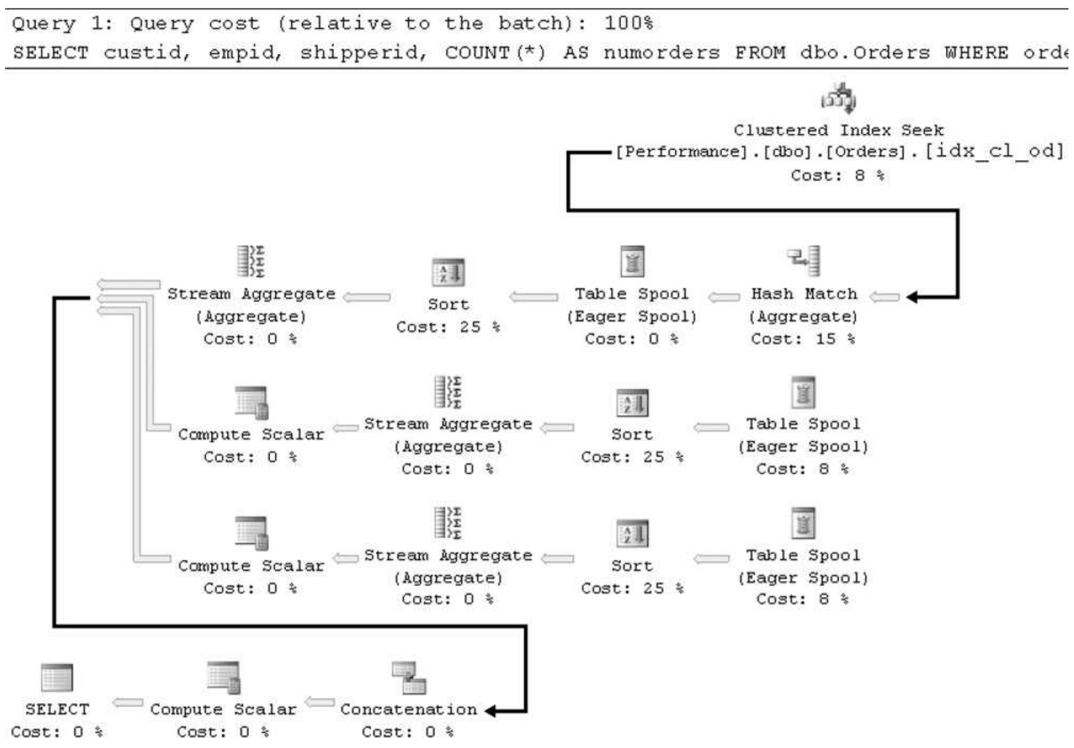


```
, empid, shipperid, COUNT(*) AS numorders
rs
te >= '20060201'
te < '20060301'
id, empid, shipperid
```

```
lative to the batch): 100%
hipperid, COUNT(*) AS numorders FROM dbo.Orders WHERE orderdate >= '2006…
```

Figure 3-8: Zooming feature in graphical showplan

Figure 3-9 shows the full execution plan for our CUBE query—that's after some graphical manipulation for clarity and to make it fit in one screen.



```
Query 1: Query cost (relative to the batch): 100%
SELECT custid, empid, shipperid, COUNT(*) AS numorders FROM dbo.Orders WHERE orde
```

Clustered Index Seek
[Performance].[dbo].[Orders].[idx_cl_od]
Cost: 8 %
Stream Aggregate (Aggregate) Cost: 0 %
Sort Cost: 25 %
Table Spool (Eager Spool) Cost: 0 %
Hash Match (Aggregate) Cost: 15 %
Compute Scalar Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %
Sort Cost: 25 %
Table Spool (Eager Spool) Cost: 8 %
Compute Scalar Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %
Sort Cost: 25 %
Table Spool (Eager Spool) Cost: 8 %
SELECT Cost: 0 %
Compute Scalar Cost: 0 %
Concatenation Cost: 0 %

Figure 3-9: Execution plan for CUBE query

I shifted the position of some of the operators and added arrows to denote the original flow. Also, I included the full object names where relevant. In the original plan, object names are truncated if they are long.

A plan is a tree of operators. Data flows from a child operator to a parent operator. The tree order of graphical plans that you get in SSMS is expressed from right to left and from top to bottom. That's typically the order in which you should analyze a plan to figure out the flow of activity. In our case, the *Clustered Index Seek* operator is the first operator that starts the flow, yielding its output to the next operator in the tree—*Hash Match (Aggregate)*—and so on.

In computer science, tree structures are typically depicted with the root node on top and leaf nodes at the bottom; and siblings' precedence is depicted from left to right. If you're accustomed to working with algorithms related to trees (or the more generic graphs), you'd probably feel comfortable with the more common representation of trees, where the execution of the operators would flow in the order depicted in Figure 3-10.

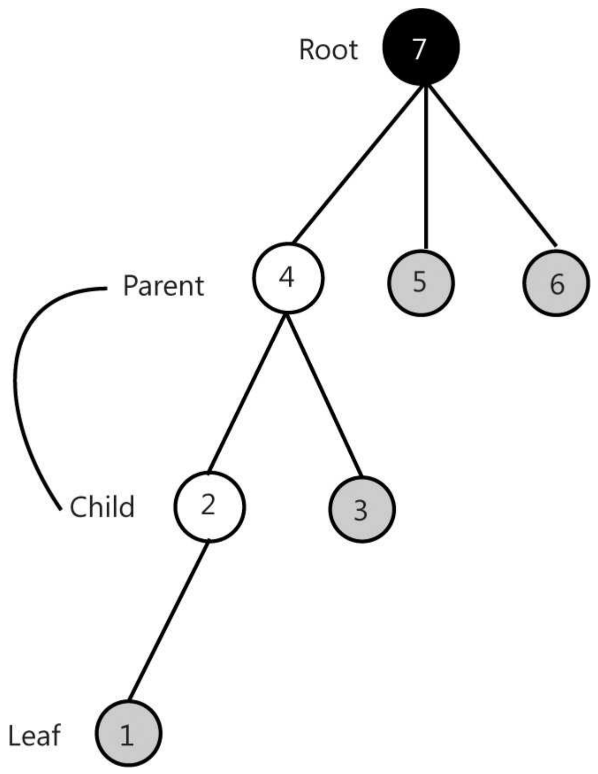


Figure 3-10: Tree

The numbers in the figure represent execution order of operators in a plan. If you feel more comfortable with the common tree representation in computer science as illustrated in Figure 3-10, you would probably appreciate the depiction shown in Figure 3-11 of our CUBE query plan.

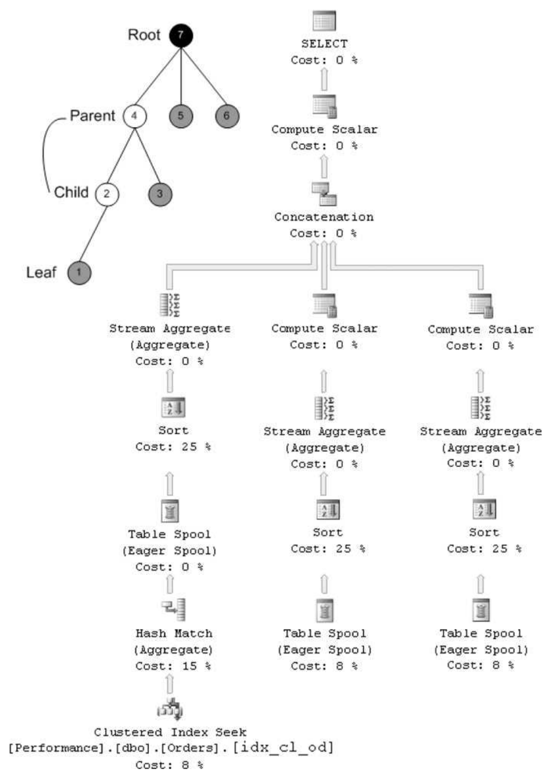
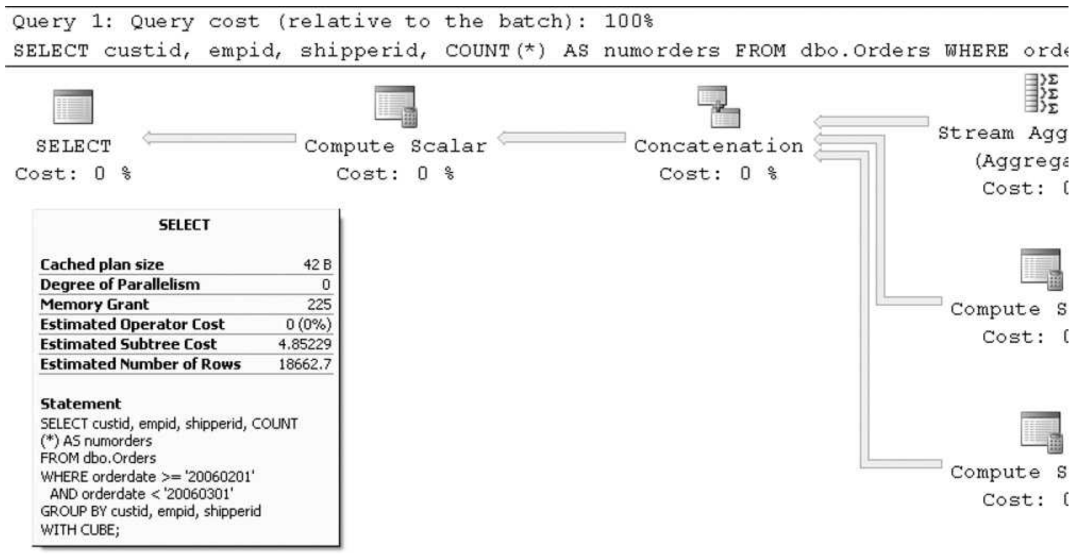


Figure 3-11: Execution plan for CUBE query (reoriented)

Logically, to produce such a plan, you need to rotate the original plan you get from SSMS 90 degrees to the left, and then flip it vertically. However, I guess that you wouldn't want to try this at home. It did take me a whole day to produce this graphic, working at the pixel level with one of my favorite tools—mspaint.exe. I did this mainly as a gesture to my mentor and friend Lubor Kollar. Lubor, this one is for you!

Go back to the original execution plan for the CUBE query shown in Figure 3-9 to examine other aspects of the plan. Notice that there's a cost percentage associated with each operator. This value is the percentage of the operator's cost out of the total cost of the query, as estimated by the optimizer. I'll explain what's behind the query's cost value shortly. You want to keep an eye especially on operators that involve high-percentage values, and focus your tuning efforts on those operators. When you place your mouse pointer over an operator, you will get a yellow information box, which I will describe shortly. One of the measures you will find there is called *Estimated Subtree Cost.* This value represents the cumulative estimated cost of the subtree, starting with the current operator (all operators in all branches leading to the current operator). The subtree cost associated with the root operator (topmost, leftmost) represents the estimated cost of the whole query, as shown in Figure 3-12.


```
Query 1: Query cost (relative to the batch): 100%
SELECT custid, empid, shipperid, COUNT(*) AS numorders FROM dbo.Orders WHERE orde
```

SELECT

Cached plan size	42 B
Degree of Parallelism	0
Memory Grant	225
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	4.85229
Estimated Number of Rows	18662.7

Statement
```
SELECT custid, empid, shipperid, COUNT
(*) AS numorders
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH CUBE;
```

Figure 3-12: Subtree cost

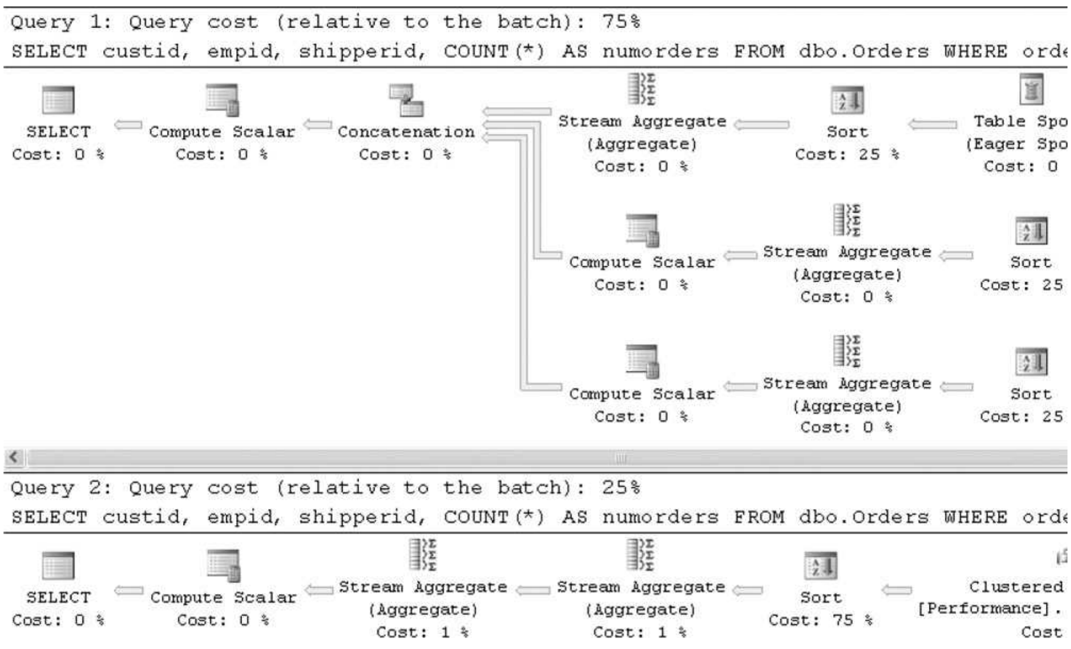
The query cost value is generated by formulas that, loosely speaking, aim at reflecting the number of seconds it would take the query to run on a test machine that SQL Server's developers used in their labs. For example, our CUBE query's estimated subtree cost is a little under 5—meaning that the formulas estimate that it would roughly take the query close to 5 seconds to run on Microsoft's test machine. Of course, that's an estimate. There are so many factors involved in the costing algorithms that even on Microsoft's original test machine you would see large variations of the actual run time of a query from its estimated cost value. Additionally, in the system you're running, your hardware and database layouts can vary significantly from the system Microsoft used for calibration. Therefore, you shouldn't expect a direct correlation between a query's subtree cost and its actual run time.

Another nice feature of the graphical execution plans is that you can easily compare the costs of multiple queries. You might want to compare the costs of different queries that produce the same result, or in some cases even queries that do slightly different things. For example, suppose you want to compare the cost of our query using the CUBE option with the same query using the ROLLUP option:

```
SELECT custid, empid, shipperid, COUNT(*)  AS numorders
FROM dbo.Orders
WHERE orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH CUBE;

SELECT custid, empid, shipperid, COUNT(*)  AS numorders
FROM dbo.Orders
WHERE   orderdate >= '20060201'
  AND orderdate < '20060301'
GROUP BY custid, empid, shipperid
WITH ROLLUP;
```

You highlight the queries that you want to compare and request a graphical execution plan (estimated or actual, as needed). In our case, you will get the plans shown in Figure 3-13.



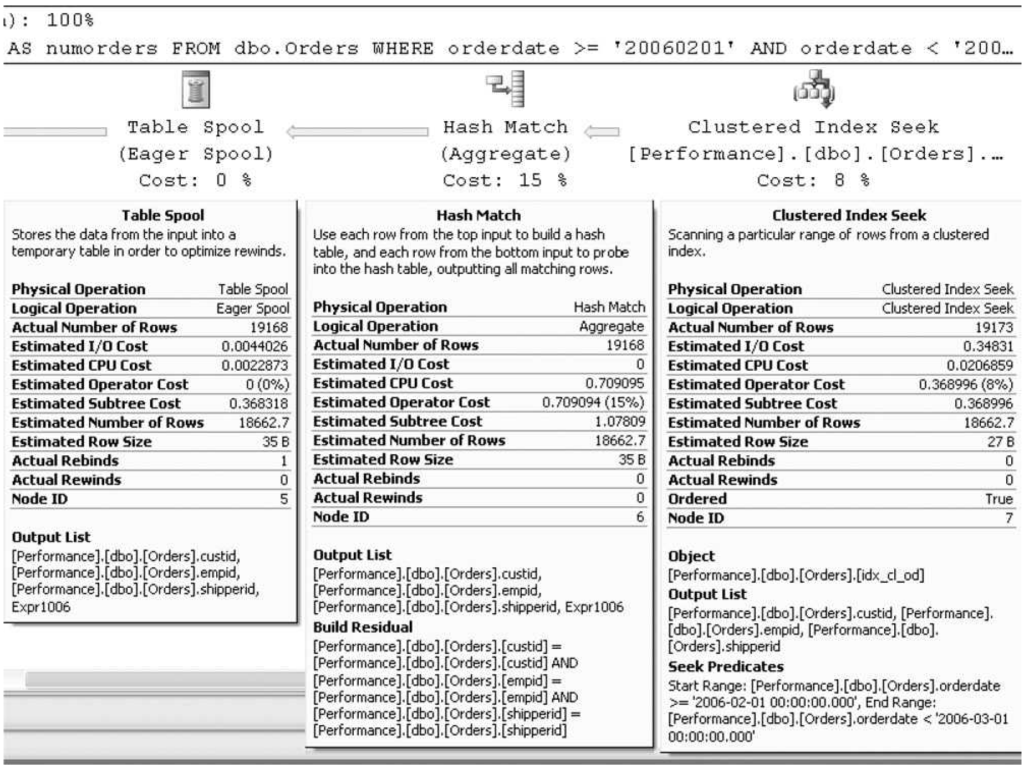
```
Query 1: Query cost (relative to the batch): 75%
SELECT custid, empid, shipperid, COUNT(*) AS numorders FROM dbo.Orders WHERE orde
```

```
Query 2: Query cost (relative to the batch): 25%
SELECT custid, empid, shipperid, COUNT(*) AS numorders FROM dbo.Orders WHERE orde
```

Figure 3-13: Comparing costs of execution plans

At the top of each plan, you will get the percentage of the estimated cost of the query out of the whole batch. For example, in our case, you can notice that the CUBE query is estimated to be three times as expensive than the ROLLUP query.

When placing your mouse pointer over an operator, you will get a yellow ToolTip box with information about the operator, as shown in Figure 3-14.



```
)): 100%
AS numorders FROM dbo.Orders WHERE orderdate >= '20060201' AND orderdate < '200…
```

Table Spool

Stores the data from the input into a temporary table in order to optimize rewinds.

Physical Operation	Table Spool
Logical Operation	Eager Spool
Actual Number of Rows	19168
Estimated I/O Cost	0.0044026
Estimated CPU Cost	0.0022873
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.368318
Estimated Number of Rows	18662.7
Estimated Row Size	35 B
Actual Rebinds	1
Actual Rewinds	0
Node ID	5

Output List
[Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].empid, [Performance].[dbo].[Orders].shipperid, Expr1006

Hash Match

Use each row from the top input to build a hash table, and each row from the bottom input to probe into the hash table, outputting all matching rows.

Physical Operation	Hash Match
Logical Operation	Aggregate
Actual Number of Rows	19168
Estimated I/O Cost	0
Estimated CPU Cost	0.709095
Estimated Operator Cost	0.709094 (15%)
Estimated Subtree Cost	1.07809
Estimated Number of Rows	18662.7
Estimated Row Size	35 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	6

Output List
[Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].empid, [Performance].[dbo].[Orders].shipperid, Expr1006

Build Residual
[Performance].[dbo].[Orders].[custid] = [Performance].[dbo].[Orders].[custid] AND [Performance].[dbo].[Orders].[empid] = [Performance].[dbo].[Orders].[empid] AND [Performance].[dbo].[Orders].[shipperid] = [Performance].[dbo].[Orders].[shipperid]

Clustered Index Seek

Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	19173
Estimated I/O Cost	0.34831
Estimated CPU Cost	0.0206859
Estimated Operator Cost	0.368996 (8%)
Estimated Subtree Cost	0.368996
Estimated Number of Rows	18662.7
Estimated Row Size	27 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	7

Object
[Performance].[dbo].[Orders].[idx_cl_od]

Output List
[Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].empid, [Performance].[dbo].[Orders].shipperid

Seek Predicates
Start Range: [Performance].[dbo].[Orders].orderdate >= '2006-02-01 00:00:00.000', End Range: [Performance].[dbo].[Orders].orderdate < '2006-03-01 00:00:00.000'

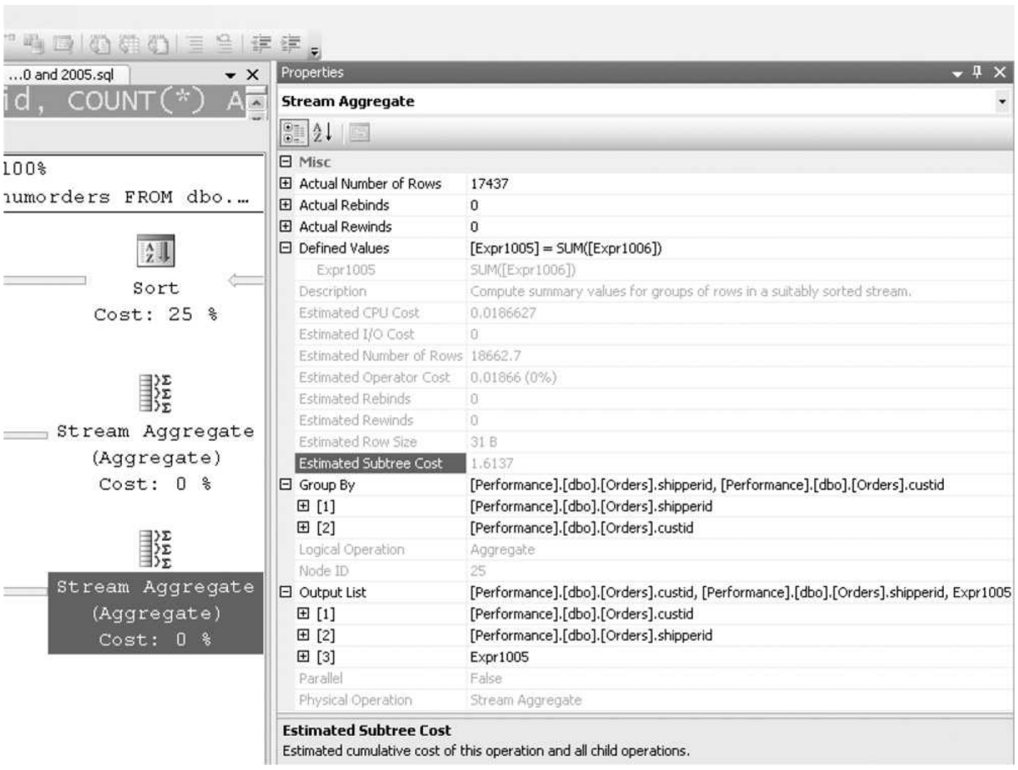
Figure 3-14: Operator info ToolTip box

The information box will give you the following information:

- The operator's name and a short description of its function.
- **Physical Operation:** The physical operation that will take place in the engine.

- **Logical Operation:** The logical operation according to Microsoft's conceptual model of query processing. For example, for a join operator you will get the join algorithm used as the physical operation (Nested Loops, Merge, Hash), and the logical join type used as the logical operation (Inner Join, Outer Join, Semi Join, and so on). When there's no logical operation associated with the operator, this measure will have the same value as shown in the physical operation.
- **Actual Number of Rows:** The actual number of rows returned from the operator (shown only for actual plans).
- **Estimated I/O Cost, and Estimated CPU Cost:** The estimated part of the operator's cost associated with that particular resource (I/O or CPU). These measures will help you identify whether the operator is I/O or CPU intensive. For example, you can see that the Clustered Index Seek operator is mainly I/O bound, while the Hash Match operator is mainly CPU bound.
- **Estimated Operator Cost:** The cost associated with the particular operator.
- **Estimated Subtree Cost:** As described earlier, the cumulative cost associated with the whole subtree up to the current node.
- **Estimated Number of Rows:** The number of rows that are estimated to be returned from this operator. In some cases, you can identify costing problems related to insufficient statistics or to other reasons by observing a discrepancy between the actual number of rows and the estimated number.
- **Estimated Row Size:** You might wonder why an actual value for this number is not shown in the actual query plan. The reason is that you might have dynamic-length attribute types in your table with rows that vary in size.
- **Actual Rebinds, and Actual Rewinds:** These measures are relevant only for operators that appear as the inner side of a Nested Loops join; otherwise, Rebinds will show 1 and Rewinds will show 0. These operators refer to the number of times that an internal *Init* method is called. The sum of the number of rebinds and rewinds should be equal to the number of rows processed on the outer side of the join. A rebind means that one or more of the correlated parameters of the join changed and the inner side must be reevaluated. A rewind means that none of the correlated parameters changed and the prior inner result set might be reused.
- **Bottom part of the info box:** Shows other aspects related to the operator, such as the associated object name, output, arguments, and so on.

In SQL Server 2005, you can get more detailed coverage of the properties of an operator in the Properties window (by pressing F4), as shown in Figure 3-15.



Stream Aggregate

Misc	
Actual Number of Rows	17437
Actual Rebinds	0
Actual Rewinds	0
Defined Values	[Expr1005] = SUM([Expr1006])
Expr1005	SUM([Expr1006])
Description	Compute summary values for groups of rows in a suitably sorted stream.
Estimated CPU Cost	0.0186627
Estimated I/O Cost	0
Estimated Number of Rows	18662.7
Estimated Operator Cost	0.01866 (0%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	31 B
Estimated Subtree Cost	1.6137
Group By	[Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].custid
[1]	[Performance].[dbo].[Orders].shipperid
[2]	[Performance].[dbo].[Orders].custid
Logical Operation	Aggregate
Node ID	25
Output List	[Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].shipperid, Expr1005
[1]	[Performance].[dbo].[Orders].custid
[2]	[Performance].[dbo].[Orders].shipperid
[3]	Expr1005
Parallel	False
Physical Operation	Stream Aggregate

Estimated Subtree Cost
Estimated cumulative cost of this operation and all child operations.

Figure 3-15: Properties window

Coverage of graphical execution plans will continue in the "Index Tuning" section when we discuss index access methods.

Textual Showplans

SQL Server also gives you tools to get an execution plan as text. For example, if you turn the SHOWPLAN_TEXT session option on, when you run a query, SQL Server will not process it. Rather, it just generates an execution plan and returns it as text. To demonstrate this session option, turn it on by running the following code:

```
SET SHOWPLAN_TEXT ON;
```

Then invoke the following query:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 280885;
```

You will get the following output:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Uniq1002],
     [Performance].[dbo].[Orders].[orderdate]))
   |--Index Seek(OBJECT:([Performance].[dbo].[Orders].[PK_Orders]),
        SEEK:([Performance].[dbo].[Orders].[orderid]=[@1])
          ORDERED FORWARD)
   |--ClusteredIndex Seek(OBJECT:(
          [Performance].[dbo].[Orders].[idx_cl_od]),
          SEEK:([Performance].[dbo].[Orders].[orderdate] =
          [Performance].[dbo].[Orders].[orderdate]
          AND[Uniq1002]=[Uniq1002]) LOOKUP ORDERED FORWARD)
```

To analyze the plan, you "read" or "follow" branches in inner levels before outer ones (bottom to top), and branches that appear in the same level from top to bottom. As you can see, you get only the operator names and their basic arguments. Run the following code to turn the session option off:

```
SET SHOWPLAN_TEXT OFF;
```

If you want more detailed information about the plan that is similar to what the graphical execution plan gives you, use the SHOWPLAN_ALL session option for an estimated plan and the STATISTICS PROFILE session option for the actual one. SHOWPLAN_ALL will produce a table result, with the information provided by SHOWPLAN_TEXT, and also the following measures: StmtText, StmtId, NodeId, Parent, PhysicalOp, LogicalOp, Argument, DefinedValues, EstimateRows, EstimateIO, EstimateCPU, AvgRowSize, TotalSubtreeCost, OutputList, Warnings, Type, Parallel, and EstimateExecutions.

To test this session option, turn it on:

```
SET SHOWPLAN_ALL ON;
```

Run the preceding query, and examine the result. When you're done, turn it off:

```
SET SHOWPLAN_ALL OFF;
```

The STATISTICS PROFILE option will produce an actual plan. The query will be run, and its output will be produced. You will also get the output returned by SHOWPLAN_ALL. In addition, you will get the attributes *Rows* and *Executes*, which hold actual values as opposed to estimated ones. To test this session option, turn it on:

```
SET STATISTICS PROFILE ON;
```

Run the preceding query, and examine the result. When you're done, turn it off:

```
SET STATISTICS PROFILE OFF;
```

XML Showplans

If you want to develop your own code that would parse and analyze execution plan information, you will find the information returned by SHOWPLAN_TEXT, SHOWPLAN_ALL, and STATISTICS PROFILE very hard to work with. SQL Server 2005 introduces two new session options that allow you to get estimated and actual execution plan information in XML format; XML data is much more convenient for an application code to parse and work with. The SHOWPLAN_XML session option will produce an XML value with the estimated plan information, and the STATISTICS XML session option will produce a value with actual plan information

To test SHOWPLAN_XML, turn it on by running the following code:

```
SET SHOWPLAN_XML ON;
```

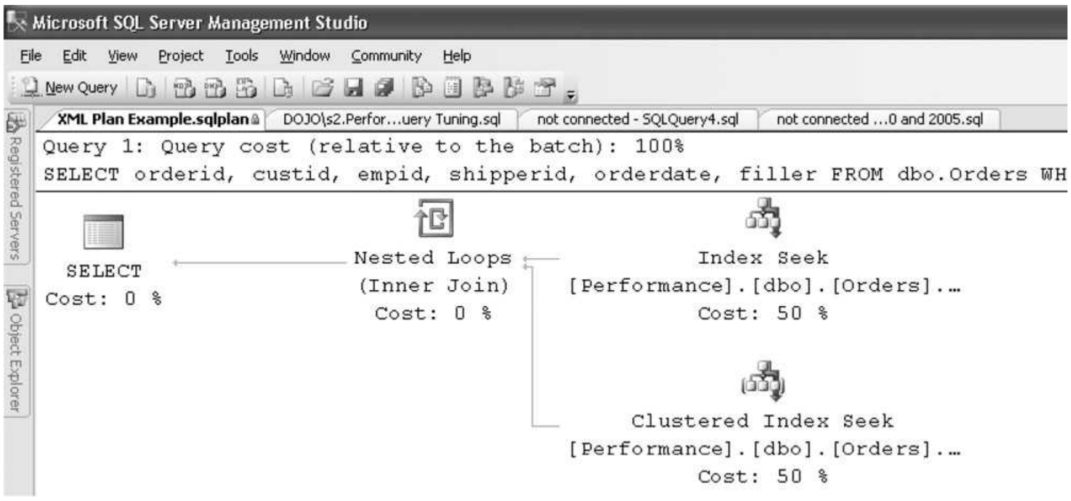
Then run the following query:

```
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 280885;
```

You will get the following XML value, shown here in abbreviated form:

```
<Show Plan XML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/
showplan" Version="1.0" Build="9.00.1399.06">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="SELECT orderid, custid, empid, shipperid, orderdate, fill
er&#xD;&#xA;FROM dbo.Orders&#xD;&#xA;WHERE orderid = 280885;" StatementId="1"  StatementCompI
d="1" StatementType="SELECT" StatementSubTreeCost="0.00657038" StatementEstRows="1" Statemen
tOptm Level="TRIVIAL">
          <StatementSetOptions   QUOTED_IDENTIFIER="false" ARITHABORT="true" CONCAT_NULL_YIELD
S_NULL="false" ANSI_NULLS="false"  ANSI_PADDING="false" ANSI_WARNINGS="false"  NUMERIC_ROUNDAB
ORT="false"/>
          <QueryPlan CachedPlanSize="14">
            <RelOp NodeId="0" PhysicalOp=" Nested Loops" LogicalOp=" InnerJoin" EstimateRows=
"1" EstimateIO="0" EstimateCPU="4.18e-006" AvgRowSize="195" EstimatedTotalSubtreeCost=
"0.00657038" Parallel="0" EstimateRebinds="0" EstimateRewinds="0">
              <OutputList>
                  <ColumnReference Database="[Performance]" Schema="[dbo]" Table="[Orders]"
Column="orderid"  />
                  <ColumnReference Database="[Performance]" Schema="[dbo]" Table="[Orders]"
Column="custid"  />
                  <ColumnReference Database="[Performance]" Schema="[dbo]" Table="[Orders]"
Column="empid"   />
                  <ColumnReference Database="[Performance]" Schema="[dbo]" Table="[Orders]"
Column="shipperid" />
                  <ColumnReference Database="[Performance]" Schema="[dbo]" Table="[Orders]"
Column="orderdate"  />
                  <ColumnReference Database="[Performance]" Schema="[dbo]" Table="[Orders]"
Column="filler"  />
              </OutputList>
              <NestedLoops Optimized="0">
                 <OuterReferences>
                   <ColumnReference Column=" Uniq1002"  />
                  <ColumnReference Database="[Performance]"  Schema="[dbo]"Table="[Orders]"
Column="orderdate"/>
                </OuterReferences>
                <RelOp NodeId="1"  PhysicalOp="Index Seek" LogicalOp="IndexSeek" EstimateRow
s="1"  EstimateIO="0.003125" EstimateCPU="0.0001581" AvgRowSize="23" EstimatedTotalSubtreeCos
t="0.0032831" Parallel="0" EstimateRebinds="0" EstimateRewinds="0">...
        </QueryPlan>
      </StmtSimple>
    </Statements>
   </Batch>
 </BatchSequence>
</Show Plan XML>
```

Note that if you save the XML value to a file with the extension *.sqlplan*, you can then open it with SSMS and get a graphical view of the execution plan, as shown in Figure 3-16.


Microsoft SQL Server Management Studio
File Edit View Project Tools Window Community Help
New Query
XML Plan Example.sqlplan
DOJO\s2.Perfor...uery Tuning.sql
not connected - SQLQuery4.sql
not connected ...0 and 2005.sql
Registered Servers
Object Explorer
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek
[Performance].[dbo].[Orders]...
Cost: 50 %
Clustered Index Seek
[Performance].[dbo].[Orders]...
Cost: 50 %

```
Query 1: Query cost (relative to the batch): 100%
SELECT orderid, custid, empid, shipperid, orderdate, filler FROM dbo.Orders WH
```

Figure 3-16: XML plan example

Run the following code to turn the session option off:

```
SET SHOWPLAN_XML OFF;
```

As I mentioned earlier, to get an XML value with information about the actual execution plan, use the STATISTICS XML session option as follows:

```
SET STATISTICS XML ON;
GO
SELECT orderid, custid, empid, shipperid, orderdate, filler
FROM dbo.Orders
WHERE orderid = 280885;
GO
SET STATISTICS XML OFF;
```

Hints

Hints allow you to override the default behavior of SQL Server in different respects, and SQL Server will comply with your request when technically possible. The term *hint* is a misnomer because it's not a kind gesture that SQL Server might or might not comply with; rather, you're forcing SQL Server to apply a certain behavior when it's technically possible. Syntactically, there are three types of hints: join hints, query hints, and table hints. Join hints are specified between the keyword representing the join type and the JOIN keyword (for example, INNER MERGE JOIN). Query hints are specified in an OPTION clause following the query itself; for example, SELECT … OPTION (OPTIMIZE FOR (@od = ‘99991231’)). Table hints are specified right after a table name or alias in a WITH clause (for example FROM dbo.Orders WITH (index = idx_unc_od_oid_i_cid_eid)).

Hints can be classified in different categories based on their functionality, including: index hints, join hints, parallelism, locking, compilation, and others. Some performance-related hints, such as forcing the usage of a certain index, have both negative and positive aspects.

On the negative side, a hint makes that particular aspect of the plan static. When data distribution in the queried tables changes, the optimizer would not consult statistics to determine whether it is worthwhile to use the index or not, because you forced it to always use it. You lose the benefit in cost-based optimization that SQL Server's optimizer gives you. On the positive side, by specifying hints you reduce the time it takes to optimize queries and, in some cases, you override inefficient choices that the optimizer occasionally makes as a result of the nature of cost-based optimization, which relies on statistics, or as a result of an optimizer bug. Make sure that you use performance-related hints in production code only after exhausting all other means, including query revisions, ensuring that statistics are up to date, and ensuring that statistics have a sufficient sampling rate, and so on.

You can find detailed information about the various supported hints in Books Online. I will use hints in several occasions in these books and explain them in context.

To conclude this section, I'd like to introduce a nifty new hint in SQL Server 2005 that you might consider to be the ultimate hint—USE PLAN. This hint allows you to provide an XML value holding complete execution plan information to force the optimizer to use the plan that you provided. Remember that you can use the SHOWPLAN_XML session option to generate an XML plan. To see a demonstration of what happens when you use this hint, first run the following code to generate the XML plan:

```
SET SHOWPLAN_XML ON;
GO
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= 2147483647;
GO
SET SHOWPLAN_XML OFF;
```

Then run the query, providing the XML plan value in the USE PLAN hint like so:

```
DECLARE @oid AS INT;
SET@oid = 1000000;

SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= @oid
OPTION (USE PLAN
N'<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/
showplan"  Version="1.0"  Build="9.00.1399.06">
 <BatchSequence>
   <Batch>
     <Statements>
       <StmtSimple StatementText="SELECT orderid, custid, empid, shipperid, orderdate&#xD;&
#xA;FROM dbo.Orders&#xD;&#xA;WHERE orderid<= 2147483647;&#xD;&#xA;"StatementId="1" Stat
ementCompId="1"  StatementType="SELECT" StatementSubTreeCost="0.00657038" StatementEstRows="1
" StatementOptmLevel="FULL" StatementOptmEarlyAbortReason="Good Enough Plan Found">
           <StatementSetOptions   QUOTED_IDENTIFIER="false" ARITHABORT="true" CONCAT_NULL_YIELD
S_NULL="false" ANSI_NULLS="false"  ANSI_PADDING="false" ANSI_WARNINGS="false"  NUMERIC_ROUNDAB
ORT="false"/>
          <QueryPlan CachedPlanSize="14">
                   <RelOp NodeId="0" PhysicalOp="Nested Loops"  LogicalOp="InnerJoin"  EstimateRows=
"1" EstimateIO="0" EstimateCPU="4.18e-006" AvgRowSize="40" EstimatedTotalSubtreeCost=
"0.00657038" Parallel="0" EstimateRebinds="0" EstimateRewinds="0">...
           <ParameterList>
           <ColumnReference Column="@1" ParameterCompiledValue="(2147483647)"/>
          </ParameterList>
         </QueryPlan>
        </StmtSimple>
       </Statements>
      </Batch>
     </BatchSequence>
    </Show Plan XML>');
```

Note that the XML value in the preceding code is shown in abbreviated form. Of course, you should specify the full-blown XML value. SQL Server 2005 also supports a new plan guide feature, which allows you to attach an XML plan to a query when you cannot or do not want to change the query's text directly by adding hints. You use the stored procedure *sp_create_plan_guide* to produce a plan guide for a query. You can find more details about it in Books Online.

SQL Server 2005 also introduces several other interesting hints, among them the RECOMPILE and OPTIMIZE FOR query hints. I'll discuss those in *Inside T-SQL Programming* as part of the discussion about stored procedure compilations and recompilations.

Traces/Profiler

The tracing capabilities of SQL Server give you extremely powerful tools for tuning and for other purposes as well. One of the great benefits tracing has over other external tools is that you get information about events that took place within the server in various components. Tracing allows you to troubleshoot performance problems, application behavior, deadlocks, audit information, and so much more. I demonstrated using traces for collecting performance workload data earlier in the book. Make sure you go over the guidelines for tracing that I provided earlier. I'll also demonstrate tracing to troubleshoot deadlocks in *Inside T-SQL Programming.* At the end of this chapter, I'll point you to additional resources that cover tracing and Profiler.

Database Engine Tuning Advisor

The Database Engine Tuning Advisor (DTA) is an enhanced tool that was formerly called Index Tuning Wizard in SQL Server 2000. DTA will give you index design recommendations based on an analysis of a workload that you give it as input. The input workload can be a trace file or table, and it can also be a script file containing T-SQL queries. One benefit of DTA is that it uses SQL Server's optimizer to make cost estimations—the same optimizer that generates execution plans for your queries. DTA generates statistics and hypothetical indexes, which it uses in its cost estimations. Among the new features in SQL Server 2005 available in DTA are partitioning recommendations for tables and indexes. Note that you can run DTA in batch mode by using the dta.exe command-line utility.

Index Tuning

This section covers index tuning, which is an important facet of query tuning. Indexes are sorting and searching structures. They reduce the need for I/O when looking for data and for sorting when certain elements in the plan need or can benefit from sorted data. While some aspects of tuning can improve performance by a modest percentage, index tuning can often improve query performance by orders of magnitude. Hence, if you're in charge of tuning, learning about indexes in depth is time well spent. Here I'll cover index tuning aspects that are relevant to these books, and at the end of the chapter I'll point you to other resources where you can find more information.

I'll start by describing table and index structures that are relevant for our discussions. Then I'll describe index access methods used by the optimizer and conclude the section by introducing an index optimization scale.

Table and Index Structures

Before delving into index access methods, you need to familiarize yourself with table and index structures. This section will describe pages and extents, heaps, clustered indexes, and nonclustered indexes.

Pages and Extents

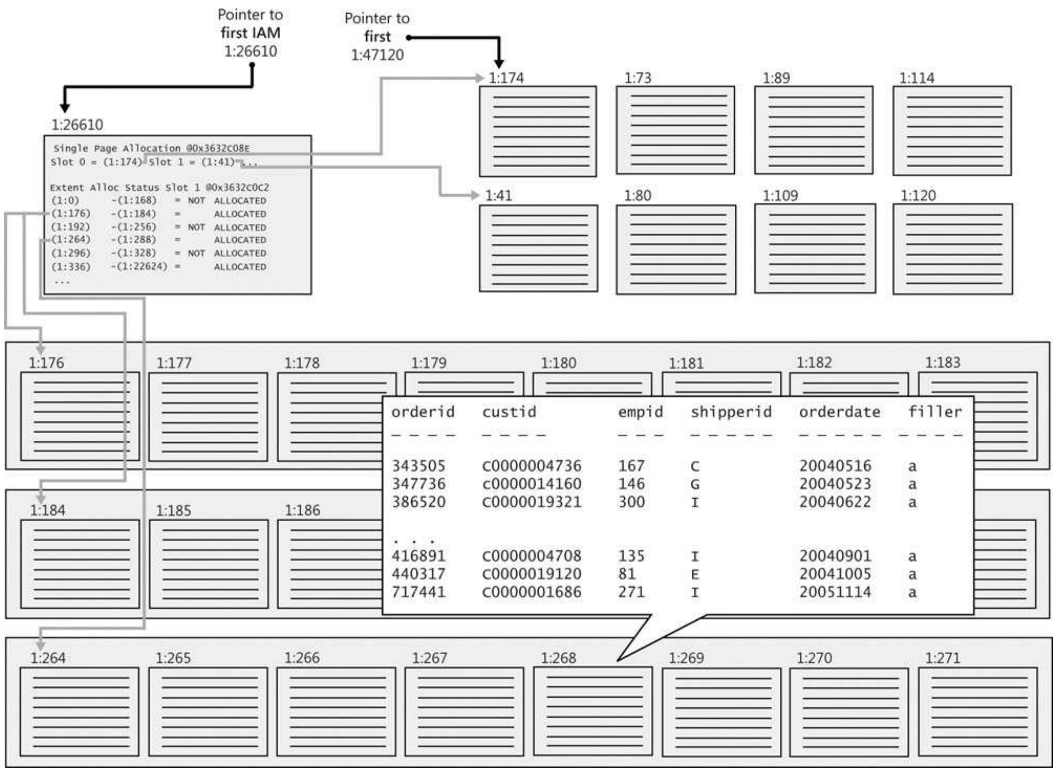
A page is an 8-KB unit where SQL Server stores data. It can contain table or index data, execution plan data, bitmaps for allocation, free space information, and so on. A page is the smallest I/O unit that SQL Server can read or write. In SQL Server 2000 and earlier, a row could not span multiple pages and was limited to 8060 bytes gross (aside from large object data). The limitation was because of the page size (8192 bytes), which was reduced by the header size (96 bytes), a pointer to the row maintained at the end of the page (2 bytes), and a few additional bytes reserved for future use. SQL Server 2005 introduces a new feature called *row-overflow data*, which relaxes the limitation on row size for tables that contain VARCHAR, NVARCHAR, VARBINARY, SQL_VARIANT, or CLR user-defined type columns. Each such column can reach up to 8000 bytes, allowing the row to span multiple pages.

Keep in mind that a page is the smallest I/O unit that SQL Server can read or write. Even if SQL Server needs to access a single row, it has to load the whole page to cache and read it from there. Queries that involve primarily data manipulation are typically bound mainly by their I/O cost. Of course, a physical read of a page is much more expensive than a logical read of a page that already resides in cache. It's hard to come up with a number that would represent the performance ratio between them, as there are several factors involved in the cost of a read, including the type of access method used, the fragmentation level of the data, and other factors as well. If you really need a ballpark number, use 1/50—that is, a logical read would very roughly be 50 times faster than a physical read. But I'd strongly advise against relying on any number as a rule of thumb.

Extents are allocation units of 8 contiguous pages. When a table or index needs more space for data, SQL Server allocates a full extent to the object. There is one exception that applies to small objects: if the object is smaller than 64 KB, SQL Server typically allocates an individual page when more space is needed, not a full extent. That page can reside within a mixed extent whose eight pages belong to different objects. Some activities of data deletion—for example, dropping a table and truncating a table—deallocate full extents. Such activities are minimally logged; therefore, they are very fast compared to the fully logged DELETE statement. Also, some read activities—such as read-ahead reads, which are typically applied for large table or index scans—can read data at the extent level. The most expensive part of an I/O operation is the movement of the disk arm, while the actual magnetic read or write operation is much less expensive; therefore, reading a page can take almost as long as reading a full extent.

Heap

A *heap* is a table that has no clustered index. The structure is called a heap because the data is not organized in any order; rather, it is laid out as a bunch of extents. Figure 3-17 illustrates how our Orders table might look like when organized as a heap.



```
Single Page Allocation @0x3632C08E
Slot 0 = (1:174) Slot 1 = (1:41) ...

Extent Alloc Status Slot 1 @0x3632C0C2
(1:0)      -(1:168)   = NOT  ALLOCATED
(1:176)    -(1:184)   =      ALLOCATED
(1:192)    -(1:256)   = NOT  ALLOCATED
(1:264)    -(1:288)   =      ALLOCATED
(1:296)    -(1:328)   = NOT  ALLOCATED
(1:336)    -(1:22624) =      ALLOCATED
 ...
```

```
orderid   custid        empid   shipperid   orderdate   filler
- - - -   - - - -       - - -   - - - - -   - - - - -   - - - -
343505    C0000004736   167     C           20040516    a
347736    C0000014160   146     G           20040523    a
386520    C0000019321   300     I           20040622    a

. . .
416891    C0000004708   135     I           20040901    a
440317    C0000019120   81      E           20041005    a
717441    C0000001686   271     I           20051114    a
```

Figure 3-17: Heap

The only structure that keeps track of the data belonging to a heap is a bitmap page (or a series of pages if needed) called the Index Allocation Map (IAM). This bitmap has pointers to the first 8 pages allocated from mixed extents, and a representative bit for each extent in a range of 4 GB in the file. The bit is 0 if the extent it represents does not belong to the object owning the IAM page, and 1 if it does. If one IAM is not enough to cover all the object's data, SQL Server will maintain a chain of IAM pages. SQL Server uses IAM pages to move through the object's data when the object needs to be scanned. SQL Server loads the object's first IAM page, and then directs the disk arm sequentially to fetch the extents by their physical order on disk.

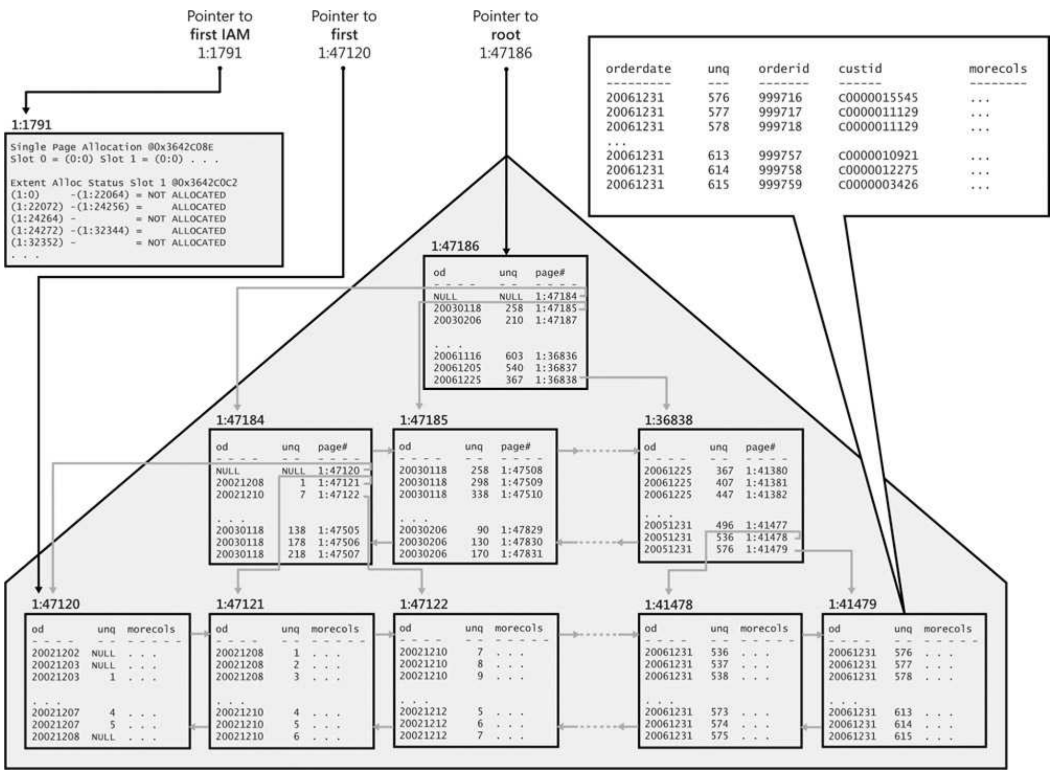
As you can see in the figure, SQL Server maintains internal pointers to the first IAM page and the first data page of a heap.

Clustered Index

All indexes in SQL Server are structured as *balanced trees.* The definition of a balanced tree (adopted from http://www.nist.gov) is: "a tree where no leaf is much farther away from the root than any other leaf."

More Info	If you're interested in the theoretical algorithmic background for balanced trees, please refer to http://www.nist.gov/dads/HTML/balancedtree.html, and to *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)* by Donald E. Knuth (Addison-Wesley Professional, 1998).

A *clustered index* is structured as a balanced tree, and it maintains all the table's data in its leaf level. The clustered index is not a copy of the data; rather, it *is* the data. I'll describe the structure of a clustered index in SQL Server through the illustration shown in Figure 3-18.



```
Single Page Allocation @0x3642C08E
Slot 0 = (0:0) Slot 1 = (0:0) . . .

Extent Alloc Status Slot 1 @0x3642C0C2
(1:0)     -(1:22064) = NOT ALLOCATED
(1:22072) -(1:24256) =     ALLOCATED
(1:24264) -          = NOT ALLOCATED
(1:24272) -(1:32344) =     ALLOCATED
(1:32352) -          = NOT ALLOCATED
. . .
```

```
orderdate   unq   orderid   custid         morecols
---------   ---   -------   ------         --------
20061231    576   999716    C0000015545    ...
20061231    577   999717    C0000011129    ...
20061231    578   999718    C0000011129    ...
...
20061231    613   999757    C0000010921    ...
20061231    614   999758    C0000012275    ...
20061231    615   999759    C0000003426    ...
```

```
1:47186
od         unq   page#
- - - -    - -   - - - -
NULL       NULL  1:47184
20030118    258  1:47185
20030206    210  1:47187

. . .
20061116    603  1:36836
20061205    540  1:36837
20061225    367  1:36838
```

```
1:47184
od         unq   page#
- - - -    - -   - - - -
NULL       NULL  1:47120
20021208      1  1:47121
20021210      7  1:47122

. . .
20030118    138  1:47505
20030118    178  1:47506
20030118    218  1:47507
```

```
1:47185
od         unq   page#
- - - -    - -   - - - -
20030118    258  1:47508
20030118    298  1:47509
20030118    338  1:47510

. . .
20030206     90  1:47829
20030206    130  1:47830
20030206    170  1:47831
```

```
1:36838
od         unq   page#
- - - -    - -   - - - -
20061225    367  1:41380
20061225    407  1:41381
20061225    447  1:41382

. . .
20051231    496  1:41477
20051231    536  1:41478
20051231    576  1:41479
```

```
1:47120
od         unq   morecols
- - - -    - -   - - - - -
20021202   NULL  . . .
20021203   NULL  . . .
20021203      1  . . .

. . .
20021207      4  . . .
20021207      5  . . .
20021208   NULL  . . .
```

```
1:47121
od         unq   morecols
- - - -    - -   - - - - -
20021208      1  . . .
20021208      2  . . .
20021208      3  . . .

. . .
20021210      4  . . .
20021210      5  . . .
20021210      6  . . .
```

```
1:47122
od         unq   morecols
- - - -    - -   - - - - -
20021210      7  . . .
20021210      8  . . .
20021210      9  . . .

. . .
20021212      5  . . .
20021212      6  . . .
20021212      7  . . .
```

```
1:41478
od         unq   morecols
- - - -    - -   - - - - -
20061231    536  . . .
20061231    537  . . .
20061231    538  . . .

. . .
20061231    573  . . .
20061231    574  . . .
20061231    575  . . .
```

```
1:41479
od         unq   morecols
- - - -    - -   - - - - -
20061231    576  . . .
20061231    577  . . .
20061231    578  . . .

. . .
20061231    613  . . .
20061231    614  . . .
20061231    615  . . .
```

Figure 3-18: Clustered table/index

The figure shows an illustration of how the Orders table might look when organized in a clustered index where the *orderdate* column is defined as the index's key column. Throughout these books, I'll refer to a table that has a clustered index as a *clustered table.* As you can see in the figure, the full data rows of the Orders table are stored in the index *leaf level.* The data rows are organized in the leaf in a sorted fashion based on the index key columns (*orderdate* in our case). A doubly linked list maintains this logical order, but note that depending on the fragmentation level of the index, the physical order of the pages on disk might not match the logical order maintained by the linked list.

Also notice that with each leaf row, the index maintains a value called a *uniquifier* (abbreviated to *unq* in the illustration). This value enumerates rows that have the same key value, and it is used together with the key value to uniquely identify rows when the index's key columns are not unique. Later, when discussing nonclustered indexes, I'll elaborate on the reasoning behind this architecture and the need to uniquely identify a row in a clustered index.

The rest of the discussion in this section is relevant to both clustered and nonclustered indexes just the same, unless explicitly stated otherwise. When SQL Server needs to perform ordered scan (or ordered partial scan) operations in the leaf level of the index, it does so by following the linked list. Note that besides the linked list, SQL Server also maintains an IAM page (or pages) to map the data stored in the index by physical order on disk. SQL Server will typically use the IAM page when it needs to perform unordered scans of the index's leaf level. The performance difference between ordered and unordered scans of the index will depend on the level of fragmentation in the index. Remember that the most expensive part of an I/O operation is the movement of the disk arm. An ordered scan in an index with no fragmentation at all will be similar in performance to an unordered scan, while the former will be substantially slower in an index with a high level of fragmentation.

On top of the leaf level of the index, the index maintains additional levels, each summarizing the level below it. Each row in a non-leaf index page points to a whole page in the level below it. The row contains two elements: the key column value of the first row in the pointed index page, and a 6-byte pointer to that page. The pointer holds the file number in the database and the page number in the file. When SQL Server builds an index, it starts from the leaf level and adds levels on top. It stops as soon as a level contains a single page, also known as the *root* page.

SQL Server always starts with the root page when it needs to navigate to a particular key at the leaf, using an access method called an *index seek*, which I'll elaborate on later in the chapter. The seek operation will "jump" from the root to the relevant page in the next level, and it will continue jumping from one level to the next until it reaches the page containing the sought key at the leaf. Remember that all leaf pages are the same distance from the root, meaning that a seek operation will cost as many page reads as the number of levels in the index. The I/O pattern of these reads is *random I/O*, as opposed to sequential I/O, because naturally the pages read by a seek operation will seldom reside next to each other.

In terms of our performance estimations, it is crucial to know what the number of levels in an index is because that number will be the cost of a seek operation in terms of page reads, and some execution plans invoke multiple seek operations repeatedly (for example, a Nested Loops join operator). For an existing index, you can get this number by invoking the INDEXPROPERTY function with the *IndexDepth* property. But for an index that you didn't create yet, you need to be familiar with the calculations that will allow you to estimate the number of levels that the index will contain.

The operands and steps required for calculating the number of levels in an index (call it *L*) are as follows (remember that these calculations apply to clustered and nonclustered indexes, unless explicitly stated otherwise):

- **The number of rows in the table (call it *num_rows):*** This is 1,000,000 in our case.
- **The average gross leaf row size (call it *leaf_row_size*)**: In a clustered index, this is actually the data row size. By "gross," I mean that you need to take the internal overhead of the row and the 2-byte pointer—stored at the end of the page—pointing to the row. The row overhead typically involves a few bytes. In our Orders table, the gross average data row size is roughly 200 bytes.
- **The average leaf page density (call it *page_density*)**: This value is the average percentage of population of leaf pages. Reasons for pages not being completely full include: data deletion, page splits caused by insertion of rows to full pages, having very large rows, and explicit requests not to populate the pages in full by specifying a *fillfactor* value when rebuilding indexes. In our case, we created a clustered index on the Orders table after populating it with the data, we did not add rows after creating the clustered index, and we did not specify a fillfactor value. Therefore, *page_density* in our case is close to 100 percent.
- **The number of rows that fit in a leaf page (call it *rows_per_leaf_page*)**: The formula to calculate this value is: *(page_size-header_size) * page_density/leaf_row_size.* Note that if you have a good estimation of *page_density*, there's no need to floor this value, as the fact that a row cannot span pages (with the aforementioned exceptions) is already accounted for in the *page_density* value. In such a case, you want to use the result number as is even if it's not an integer. On the other hand, if you just estimate that *page_density* will be close to 100 percent, as it is in our case, omit the *page_density* operand from the calculation and floor the result. In our case, *rows_per_leaf_page* amount to *floor((8192-96)/200) = 40.*
- **The number of pages maintained in the leaf (call it *num_leaf_pages*)**: This is a simple formula: *num_rows/rows_per_leaf_page.* In our case, it amounts to *1,000,000/40 = 25,000.*
- **The average gross non-leaf row size (call it *non_leaf_row_size*)**: A non-leaf row contains the key columns of the index (in our case, only *orderdate*, which is 8 bytes); the 4-byte *uniquifier* (which exists only in a clustered index that is not unique); the page pointer, which is 6 bytes; a few additional bytes of internal overhead, which total 5 bytes in our case; and the row offset pointer at the end of the page, which is 2 bytes. In our case, the gross non-leaf row size is 25 bytes.
- **The number of rows that can fit in a non-leaf page (call it *rows_per_non_leaf_page*)**: The formula to calculate this value is similar to calculating *rows_per_leaf_page.* For the sake of simplicity, I'll ignore the non-leaf page density factor, and calculate the value as *floor((page_size-header_size)/non_leaf_row_size)*, which in our case amounts to *floor((8192-96)/25) = 323.*
- **The number of levels above the leaf (call it *L-1*)**: This value is calculated with the following formula: *ceiling* $(\log_{rows\_per\_non\_leaf\_page}(num\_leaf\_pages))$. In our case, *L-1* amounts to $ceiling(\log_{323}(25000)) = 2.$ Obviously, you simply need to add 1 to get *L*, which in our case is 3.

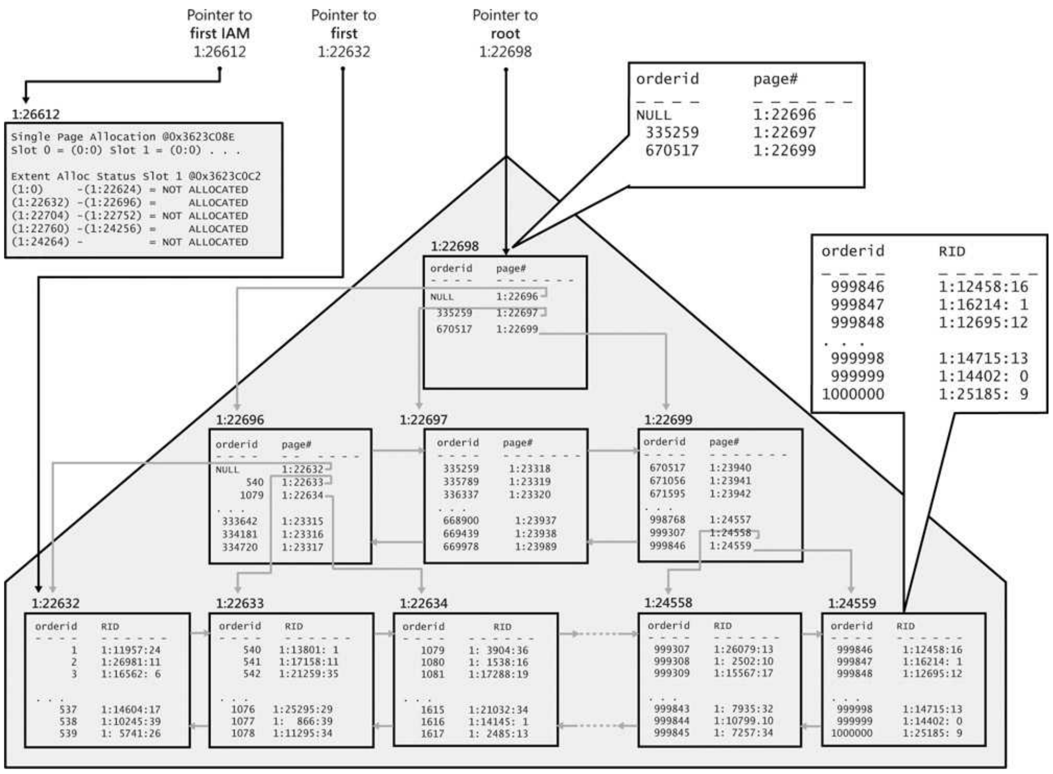
This exercise leads me to a very important point that I will rely on in my performance discussions. You can play with the formula and see that with up to about several thousand rows, our index will have 2 levels. Three levels would have up to about 4,000,000 rows, and 4 levels would have up to about 4,000,000,000 rows. With nonclustered indexes, the formulas are identical, it's just that you can fit more rows in each leaf page, as I will describe later. So with nonclustered indexes, the upper bound for each number of levels covers even more rows in the table. The point is that in our table all indexes have 3 levels, which is the cost you have to consider in your performance estimation when measuring the cost of a seek operation.

And in general, with small tables most indexes will typically have up to 2 levels, and with large tables, they will typically have 3 or 4 levels, unless the total size of the index keys is large. Keep these numbers in mind for our later discussions.

Nonclustered Index on a Heap

A nonclustered index is also structured as a balanced tree, and in many respects is similar to a clustered index. The only difference is that a leaf row in a nonclustered index contains only the index key columns and a *row locator* value pointing to a particular data row. The content of the row locator depends on whether the table is a heap or a clustered table. This section describes nonclustered indexes on a heap, and the following section will describe nonclustered indexes on a clustered table.

Figure 3-19 illustrates the nonclustered index created by our primary key constraint (*PK_Orders*) defining the *orderid* column as the key column.



Pointer to first IAM 1:26612
Pointer to first 1:22632
Pointer to root 1:22698
1:26612
Single Page Allocation @0x3623C08E
Slot 0 = (0:0) Slot 1 = (0:0) . . .
Extent Alloc Status Slot 1 @0x3623C0C2
(1:0) -(1:22624) = NOT ALLOCATED
(1:22632) -(1:22696) = ALLOCATED
(1:22704) -(1:22752) = NOT ALLOCATED
(1:22760) -(1:24256) = ALLOCATED
(1:24264) - = NOT ALLOCATED
orderid page#
NULL 1:22696
335259 1:22697
670517 1:22699
1:22698
1:22696
1:22697
1:22699
orderid RID
999846 1:12458:16
999847 1:16214: 1
999848 1:12695:12
999998 1:14715:13
999999 1:14402: 0
1000000 1:25185: 9
1:22632
1:22633
1:22634
1:24558
1:24559

Figure 3-19: Nonclustered index on a heap

The row locator used by a nonclustered index leaf row to point to a data row is an 8-byte physical pointer called *RID.* It consists of the file number in the database, the target page number in the file, and the row number in the target page (zero-based). When looking for a particular data row through the index, SQL Server will have to follow the seek operation with an *RID lookup* operation, which translates to reading the page that contains the data row. Therefore, the cost of an RID lookup is one page read. For a single lookup or a very small number of lookups the cost is not high, but for a large number of lookups the cost can be very high because SQL Server ends up reading one whole page per sought row. For range queries that use a nonclustered index, and a series of lookups—one per qualifying key—the cumulative cost of the lookup operations typically makes up the bulk of the cost of the query. I'll demonstrate this point in the "Index Access Methods" section. As for the cost of a seek operation, remember that the formulas I provided earlier are just as relevant to nonclustered indexes. It's just that the *leaf_row_size* is smaller, and therefore the *rows_per_leaf_page* will be higher. But the formulas are the same.

Nonclustered Index on a Clustered Table

As of SQL Server 7.0, nonclustered indexes created on a clustered table are architected differently than on a heap. The only difference is that the row locator in a nonclustered index created on a clustered table is a value called a *clustering key*, as opposed to being an RID. The clustering key consists of the values of the clustered index keys from the pointed row, and the *uniquifier* (if present). The idea is to point to a row "logically" as opposed to "physically." This architecture was designed mainly for OLTP systems, where clustered indexes often suffer from many page splits upon data insertions. In a page split, half the rows from the split page are physically moved to a newly allocated page. If nonclustered indexes kept physical pointers to rows, all pointers to the data rows that moved would have to be changed to reflect their new physical locations—and that's true for all relevant pointers in all nonclustered indexes. Instead, SQL Server maintains logical pointers that don't change when data rows physically move.

Figure 3-20 illustrates what the *PK_Orders* nonclustered index might look like; the index is defined with the *orderid* as the key column, and the Orders table has a clustered index defined with the *orderdate* as the key column.

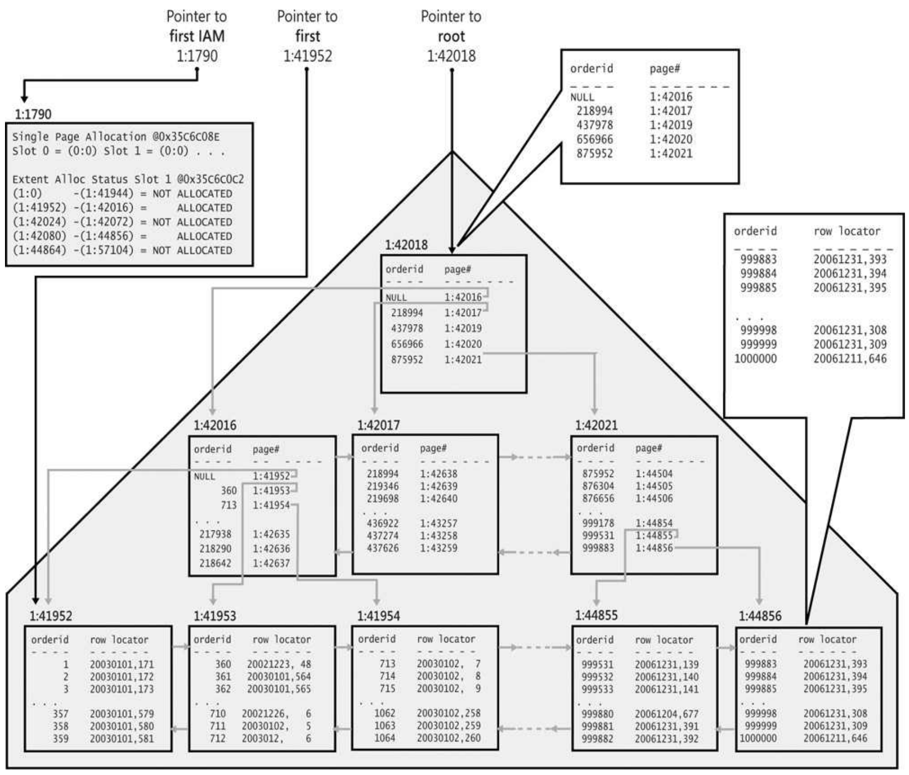


Figure 3-20: Nonclustered index on a clustered table

A seek operation looking for a particular key in the nonclustered index (some *orderid* value) will end up reaching the relevant leaf row and have access to the row locator. The row locator in this case is the clustering key of the pointed row. To actually grab the pointed row, a lookup operation will need to perform a whole seek within the clustered index based on the acquired clustering key. I will demonstrate this access method later in the chapter. The cost of each lookup operation here (in terms of the number of page reads) is as high as the number of levels in the clustered index (3 in our case). That's compared to a single page read for an RID lookup when the table is a heap. Of course, with range queries that use a nonclustered index and a series of lookups, the ratio between the number of logical reads in a heap case and a clustered table case will be close to *1: L*, where *L* is the number of levels in the clustered index.

Before you worry too much about this point and remove all clustered indexes from your tables, keep in mind that with all lookups going through the clustered index, the non-leaf levels of the clustered index will typically reside in cache. Typically, most of the physical reads in the clustered index will be against the leaf level. Therefore, the additional cost of lookups against a clustered table compared to a heap is usually a small portion of the total query cost. Now that the background information about table and index structures has been covered, the next section will describe index access methods.

Index Access Methods

This section provides a technical description of the various index access methods; it is designed to be used as a reference for discussions in these books involving analysis of execution plans. Later in this chapter, I'll describe an index optimization scale that demonstrates how you can put this knowledge into action.

If you want to follow the examples in this section, rerun the code in Listing 3-1 to re-create the sample tables in our Performance database along with all the indexes. I'll be discussing some access methods to use against the Orders table that are structured as a heap and some that are structured as a clustered table. Therefore, I'd also suggest that you run the code in Listing 3-1 against another database (say, Performance2), after renaming the database name in the script accordingly, and commenting out the statement that creates the clustered index on Orders. When I discuss an access method involving a clustered table, run the code against the Performance database. When the discussion is about heaps, run it against Performance2. Also remember that Listing 3-1 uses randomization to populate the customer IDs, employee IDs, shipper Ids, and order dates in the Orders table. This means that your results will probably slightly differ from mine.

Table Scan/Unordered Clustered Index Scan

A *table scan* or an *unordered clustered index scan* simply involves a sequential scan of all data pages belonging to the table. The following query against the Orders table structured as a heap would require a table scan:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders;
```

Figure 3-21 shows an illustration of this access method, and Figure 3-22 shows the graphical execution plan you would get for this query.

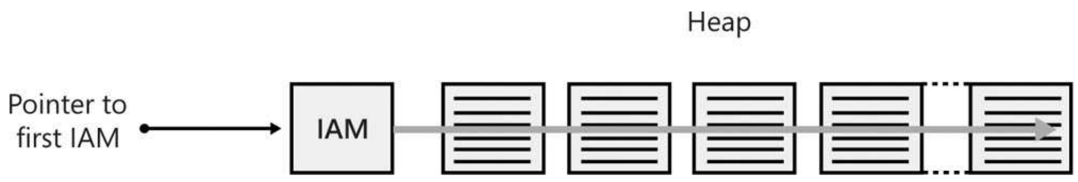


Figure 3-21: Table scan

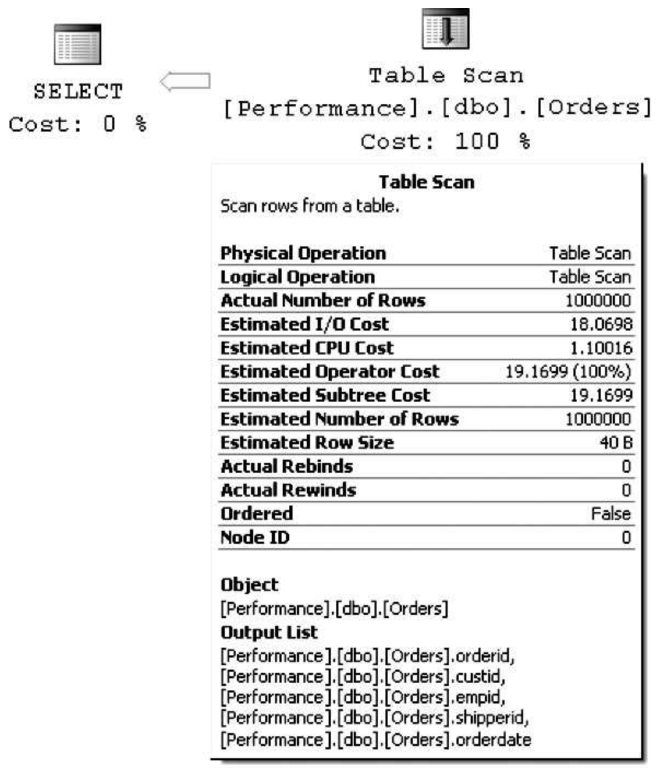


Table Scan
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Number of Rows	1000000
Estimated I/O Cost	18.0698
Estimated CPU Cost	1.10016
Estimated Operator Cost	19.1699 (100%)
Estimated Subtree Cost	19.1699
Estimated Number of Rows	1000000
Estimated Row Size	40 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Object
[Performance].[dbo].[Orders]
Output List
[Performance].[dbo].[Orders].orderid,
[Performance].[dbo].[Orders].custid,
[Performance].[dbo].[Orders].empid,
[Performance].[dbo].[Orders].shipperid,
[Performance].[dbo].[Orders].orderdate

Figure 3-22: Table scan (execution plan)

SQL Server will use the table's IAM pages to direct the disk arm to scan the extents belonging to the table sequentially by their physical order on disk. The number of logical reads should be similar to the number of pages the table consumes (around 25,000 in our case). Note that in such scans SQL Server typically uses a very efficient read-ahead strategy that can read the data in larger chunks than 8 KB. When I ran this query on my system, I got the following performance measures from STATISTICS IO, STATISTICS TIME, and the execution plan:

- Logical reads: 24391
- Physical reads: 0
- Read-ahead reads: 24408

- CPU time: 971 ms
- Elapsed time: 20265 ms
- Estimated subtree cost: 19.1699

Of course, the run times I got are not an indication of the run times you would get in an average production system. But I wanted to show them for illustration and comparison purposes.

If the table has a clustered index, the access method that will be applied will be an unordered clustered index scan, as illustrated in Figure 3-23. It will produce the execution plan shown in Figure 3-24.

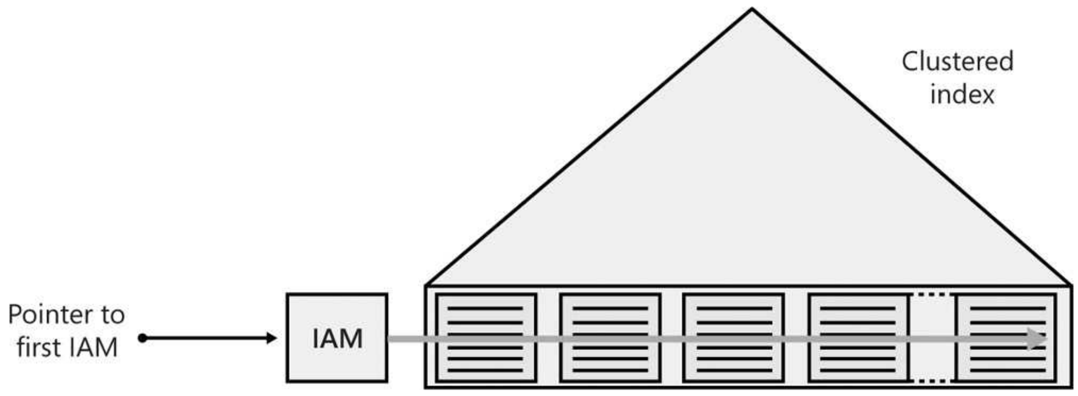
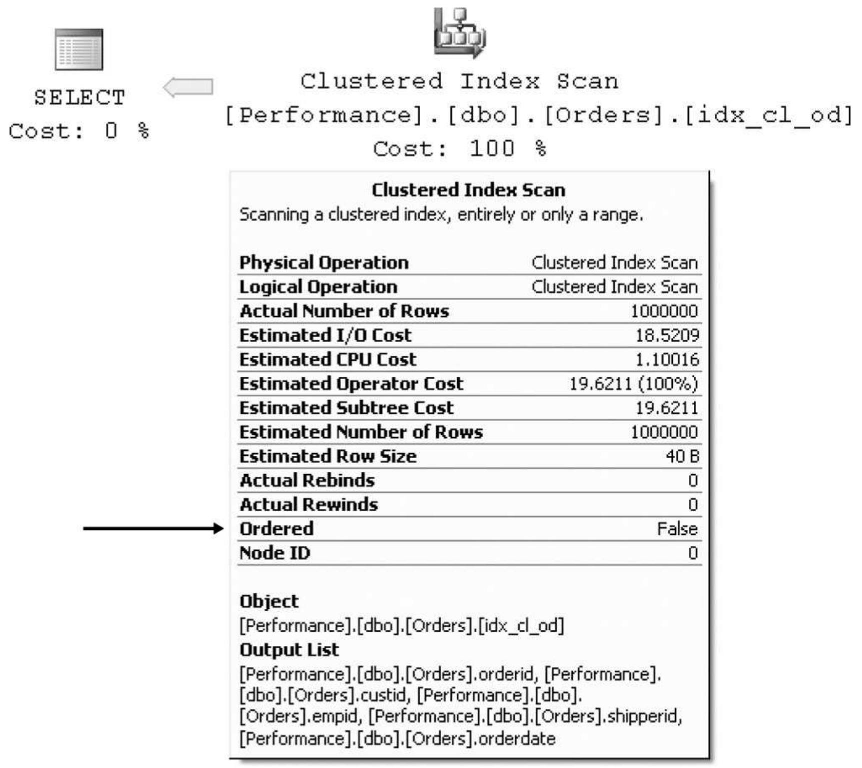


Figure 3-23: Unordered clustered index scan



Clustered Index Scan
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	1000000
Estimated I/O Cost	18.5209
Estimated CPU Cost	1.10016
Estimated Operator Cost	19.6211 (100%)
Estimated Subtree Cost	19.6211
Estimated Number of Rows	1000000
Estimated Row Size	40 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Object
[Performance].[dbo].[Orders].[idx_cl_od]
Output List
[Performance].[dbo].[Orders].orderid, [Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].empid, [Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].orderdate

Figure 3-24: Unordered clustered index scan (execution plan)

Note that even though the execution plan shows a clustered index scan, the activity is no different than a table scan, and throughout the book I will often refer to it simply as a table scan. As shown in the illustration, here SQL Server will also use the index's IAM pages to scan the data sequentially. The information box of the Clustered Index Scan operator tells you that the scan was not ordered, meaning that the access method did not rely on the linked list that maintains the logical order of the index. Of course, you did get a scan of the pages by their physical order on disk. Here are the performance measures I got for this query:

- Logical reads: 25080
- Physical reads: 1
- Read-ahead reads: 25071
- CPU time: 941 ms
- Elapsed time: 22122 ms
- Estimated subtree cost: 19.6211

Unordered Covering Nonclustered Index Scan

An *unordered covering nonclustered index scan* is similar in concept to an unordered clustered index scan. The concept of a *covering index* means that a nonclustered index contains all columns specified in a query. In other words, a covering index is not an index with special properties; rather, it becomes a covering index with respect to a particular query. SQL Server can find all the data it needs to satisfy the query by accessing solely the index data, without the need to access the full data rows. Other than that, the access method is the same as an unordered clustered index scan; only obviously, the leaf level of the covering nonclustered index contains fewer pages than the leaf of the clustered index, because the row size is smaller and more rows fit in each page. I explained earlier how to calculate the number of pages in the leaf level of an index (clustered or nonclustered).

As an example for this access method, the following query requests all *orderid* values from the Orders table:

```
SELECT orderid
FROM dbo.Orders;
```

Our Orders table has a nonclustered index on the *orderid* column (*PK_Orders*), meaning that all the table's order IDs reside in the index's leaf level. The index covers our query. Figure 3-25 illustrates this access method, and Figure 3-26 shows the graphical execution plan you would get for this query.

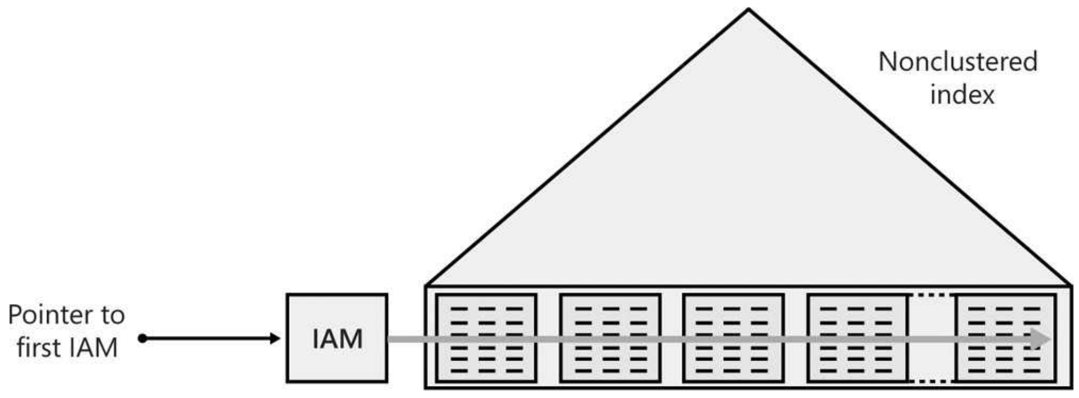
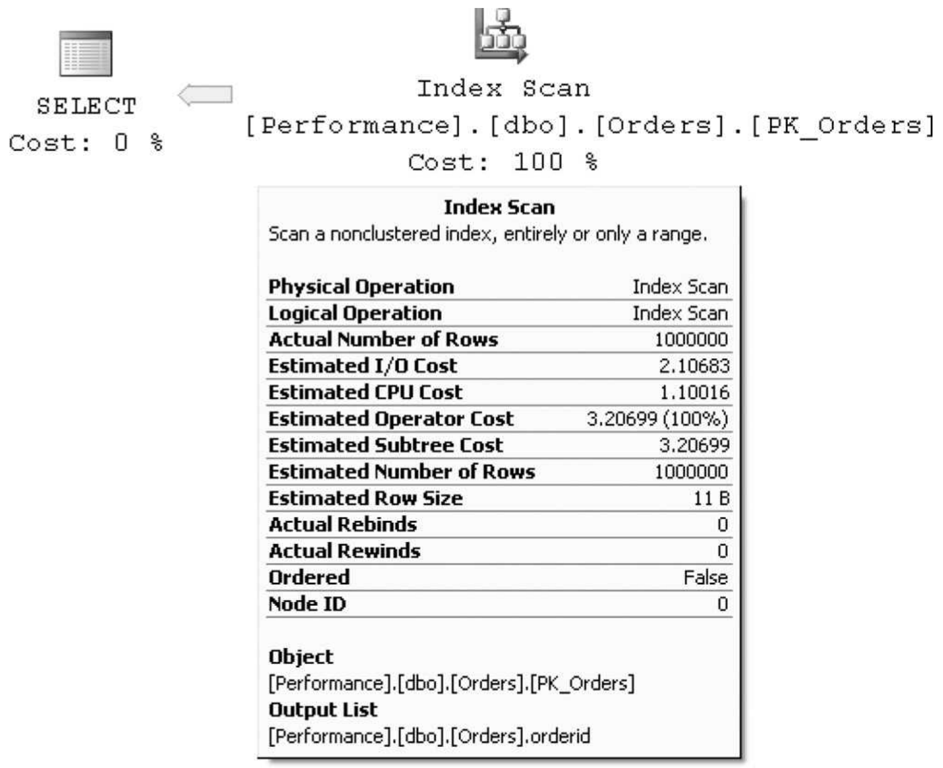


Figure 3-25: Unordered covering nonclustered index scan



Index Scan
Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Number of Rows	1000000
Estimated I/O Cost	2.10683
Estimated CPU Cost	1.10016
Estimated Operator Cost	3.20699 (100%)
Estimated Subtree Cost	3.20699
Estimated Number of Rows	1000000
Estimated Row Size	11 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Object
[Performance].[dbo].[Orders].[PK_Orders]
Output List
[Performance].[dbo].[Orders].orderid

Figure 3-26: Unordered covering nonclustered index scan (execution plan)

The leaf level of the *PK_Orders* index contains fewer than 3000 pages, compared to the 25,000 data pages in the table. Here are the performance measures I got for this query:

- Logical reads: 2848
- Physical reads: 1
- Read-ahead reads: 2841
- CPU time: 340 ms
- Elapsed time: 12071 ms
- Estimated subtree cost: 3.20699

Ordered Clustered Index Scan

An *ordered clustered index scan* is a full scan of the leaf level of the clustered index following the linked list. For example, the following query, which requests all orders sorted by *orderdate*, will get such an access method in its plan:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
ORDER BY orderdate;
```

You can find an illustration of this access method in Figure 3-27, and the execution plan for this query in Figure 3-28.

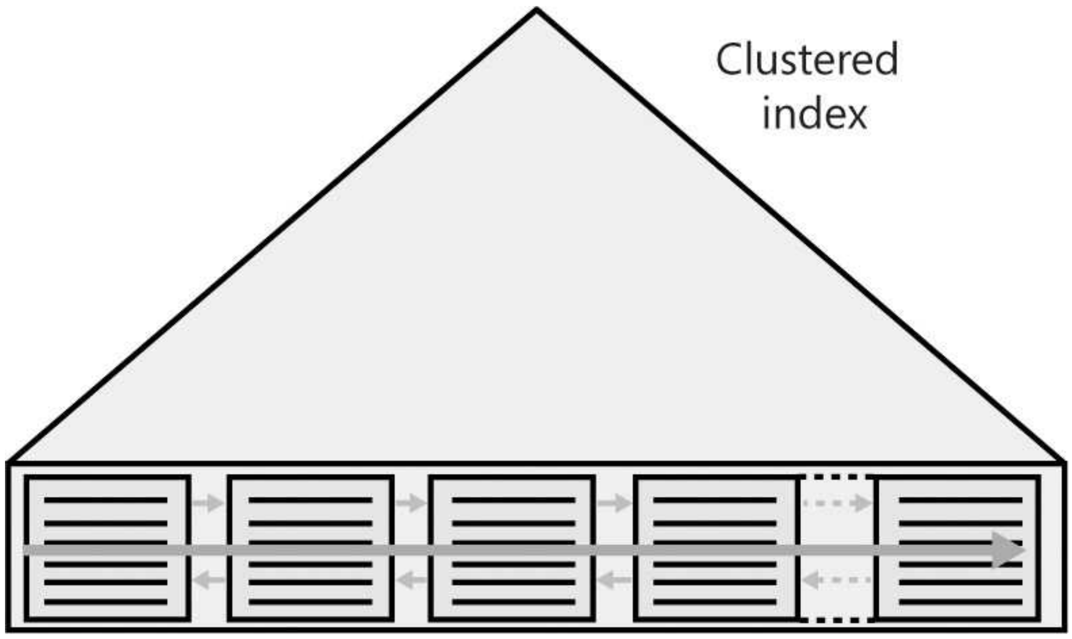
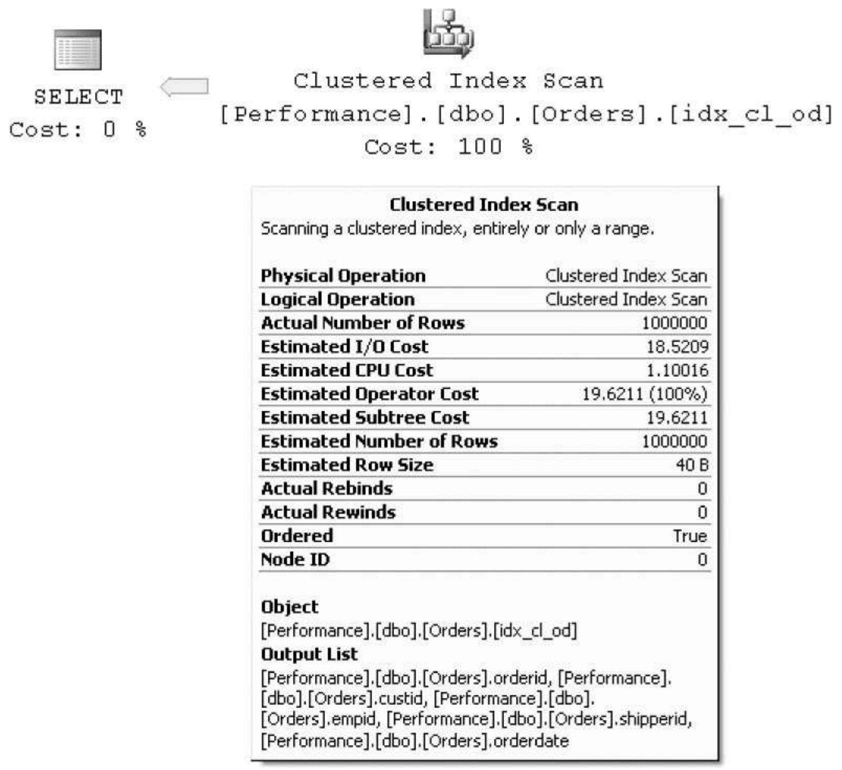


Figure 3-27: Ordered clustered index scan



Clustered Index Scan
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	1000000
Estimated I/O Cost	18.5209
Estimated CPU Cost	1.10016
Estimated Operator Cost	19.6211 (100%)
Estimated Subtree Cost	19.6211
Estimated Number of Rows	1000000
Estimated Row Size	40 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0

Object
[Performance].[dbo].[Orders].[idx_cl_od]
Output List
[Performance].[dbo].[Orders].orderid, [Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].empid, [Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].orderdate

Figure 3-28: Ordered clustered index scan (execution plan)

Note that unlike an unordered scan of an index, the performance of an ordered scan will depend on the fragmentation level of the index—that is, the percentage of the out-of-order pages in the leaf level of the index with respect to the total number of pages. An out-of-order page is a page that appears logically after a certain page according to the linked list, but physically before it. With no fragmentation at all, the performance of an ordered scan of an index should be very close to the performance of an ordered scan, because both will end up reading the data physically in a sequential manner. However, as the fragmentation level grows higher, the performance difference will be more substantial, in favor of the unordered scan, of course. The natural deductions are that you shouldn't request the data sorted if you don't need it sorted, and that you should resolve fragmentation issues in indexes that incur large ordered scans. I'll elaborate on fragmentation and its treatment later. Here are the performance measures that I got for this query:

- Logical reads: 25080

- Physical reads: 1
- Read-ahead reads: 25071
- CPU time: 1191 ms
- Elapsed time: 22263 ms
- Estimated subtree cost: 19.6211

Note that the optimizer is not limited to ordered-forward activities. Remember that the linked list is a doubly linked list, where each page contains both a *next* and a *previous* pointer. Had you requested a descending sort order, you would have still gotten an ordered index scan, only ordered backwards (from tail to head) instead of ordered forward (from head to tail). SQL Server also supports descending indexes as of version 2000, but these are not needed in simple cases like getting descending sort orders. Rather, descending indexes are valuable when you create an index on multiple key columns that have opposite directions in their sort requirements—for example, sorting by col1, col2 DESC.

Ordered Covering Nonclustered Index Scan

An *ordered covering nonclustered index scan* is similar in concept to an ordered clustered index scan, with the former performing the access method in a nonclustered index—typically when covering a query. The cost is of course lower than a clustered index scan because fewer pages are involved. For example, the *PK_Orders* index on our clustered Orders table happens to cover the following query, even though it might not seem so at first glance:

```
SELECT orderid, orderdate
FROM dbo.Orders
ORDER BY orderid;
```

Keep in mind that on a clustered table, nonclustered indexes will use clustering keys as row locators. In our case, the clustering keys contain the *orderdate* values, which can be used for covering purposes as well. Also, the first (and, in our case, the only) key column in the nonclustered index is the *orderid* column, which is the column specified in the ORDER BY clause of the query; therefore, an ordered index scan is a natural access method for the optimizer to choose.

Figure 3-29 illustrates this access method, and Figure 3-30 shows the query's execution plan.

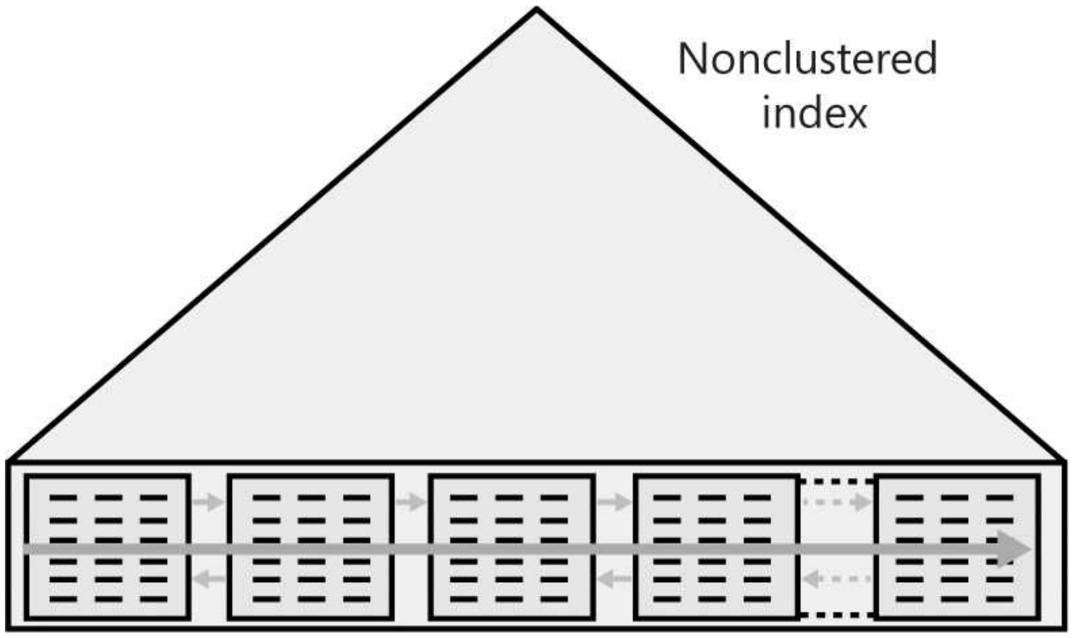
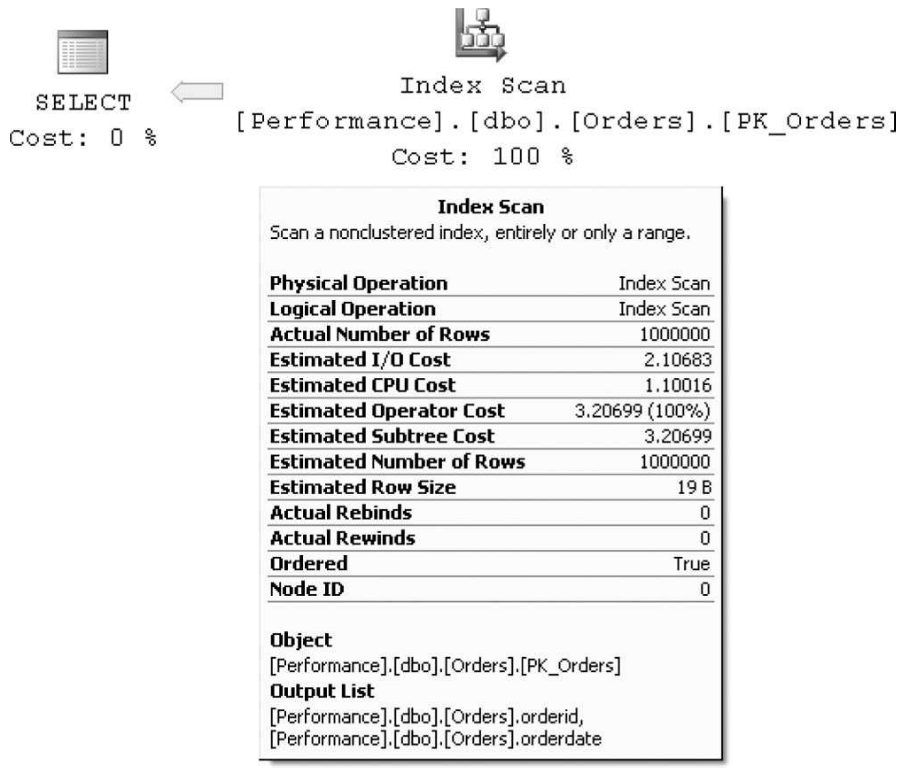


Figure 3-29: Ordered covering nonclustered index scan



Index Scan
Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Number of Rows	1000000
Estimated I/O Cost	2.10683
Estimated CPU Cost	1.10016
Estimated Operator Cost	3.20699 (100%)
Estimated Subtree Cost	3.20699
Estimated Number of Rows	1000000
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0

Object
[Performance].[dbo].[Orders].[PK_Orders]
Output List
[Performance].[dbo].[Orders].orderid,
[Performance].[dbo].[Orders].orderdate

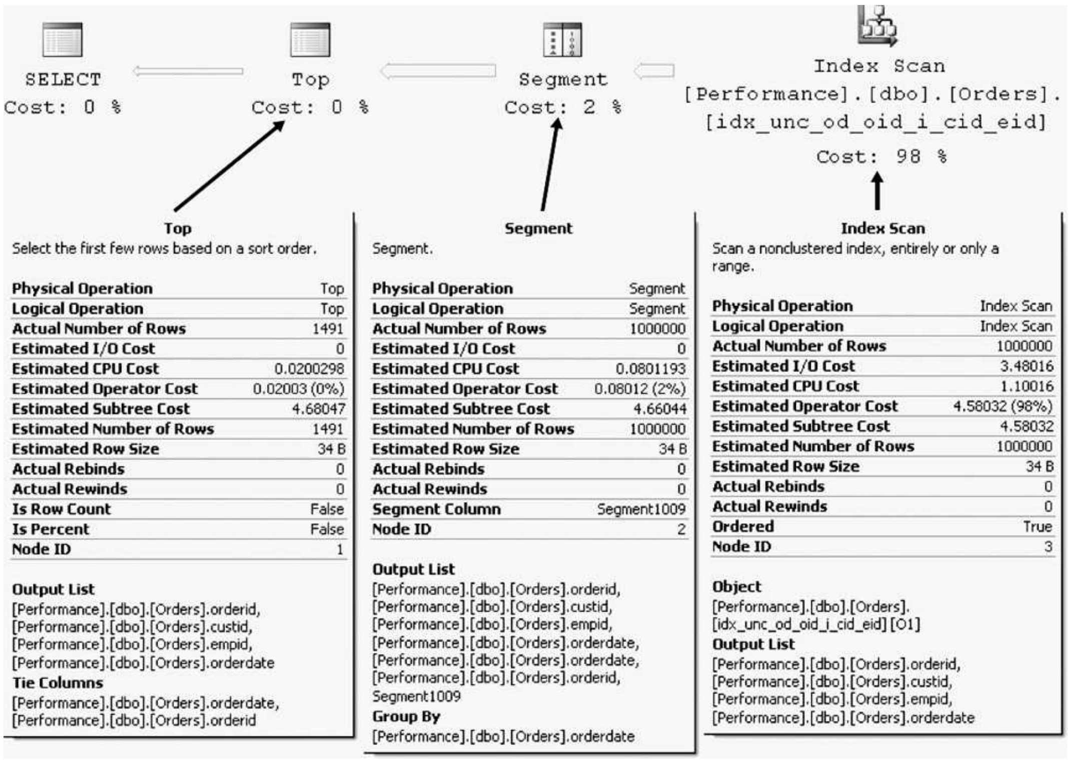
Figure 3-30: Ordered covering nonclustered index scan (execution plan 1)

Notice in the plan that the *Ordered* measure for the *Index Scan* operator in the yellow information box shows *True.*

Here are the performance measures that I got for this query:

- Logical reads: 2848
- Physical reads: 2
- Read-ahead reads: 2841
- CPU time: 370 ms
- Elapsed time: 13582 ms
- Estimated subtree cost: 3.20699

An ordered index scan is not used only when you explicitly request the data sorted; rather, it is also used when the plan uses an operator that can benefit from sorted input data. For example, check out the execution plan shown in Figure 3-31 for the following query:



Top

Select the first few rows based on a sort order.

Physical Operation	Top
Logical Operation	Top
Actual Number of Rows	1491
Estimated I/O Cost	0
Estimated CPU Cost	0.0200298
Estimated Operator Cost	0.02003 (0%)
Estimated Subtree Cost	4.68047
Estimated Number of Rows	1491
Estimated Row Size	34 B
Actual Rebinds	0
Actual Rewinds	0
Is Row Count	False
Is Percent	False
Node ID	1

Output List
[Performance].[dbo].[Orders].orderid,
[Performance].[dbo].[Orders].custid,
[Performance].[dbo].[Orders].empid,
[Performance].[dbo].[Orders].orderdate

Tie Columns
[Performance].[dbo].[Orders].orderdate,
[Performance].[dbo].[Orders].orderid

Segment

Segment.

Physical Operation	Segment
Logical Operation	Segment
Actual Number of Rows	1000000
Estimated I/O Cost	0
Estimated CPU Cost	0.0801193
Estimated Operator Cost	0.08012 (2%)
Estimated Subtree Cost	4.66044
Estimated Number of Rows	1000000
Estimated Row Size	34 B
Actual Rebinds	0
Actual Rewinds	0
Segment Column	Segment1009
Node ID	2

Output List
[Performance].[dbo].[Orders].orderid,
[Performance].[dbo].[Orders].custid,
[Performance].[dbo].[Orders].empid,
[Performance].[dbo].[Orders].orderdate,
[Performance].[dbo].[Orders].orderdate,
[Performance].[dbo].[Orders].orderid,
Segment1009

Group By
[Performance].[dbo].[Orders].orderdate

Index Scan

Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Number of Rows	1000000
Estimated I/O Cost	3.48016
Estimated CPU Cost	1.10016
Estimated Operator Cost	4.58032 (98%)
Estimated Subtree Cost	4.58032
Estimated Number of Rows	1000000
Estimated Row Size	34 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3

Object
[Performance].[dbo].[Orders].
[idx_unc_od_oid_i_cid_eid] [O1]

Output List
[Performance].[dbo].[Orders].orderid,
[Performance].[dbo].[Orders].custid,
[Performance].[dbo].[Orders].empid,
[Performance].[dbo].[Orders].orderdate

Figure 3-31: Ordered covering nonclustered index scan (execution plan 2)

```
SELECT orderid, custid, empid, orderdate
FROM dbo.Orders AS O1
WHERE orderid =
  (SELECT MAX(orderid)
   FROM dbo.Orders AS O2
   WHERE O2.orderdate = O1.orderdate);
```

The *Segment* operator arranges the data in groups and emits a group at a time to the next operator (*Top* in our case). Our query requests the orders with the maximum *orderid* per *orderdate.* Fortunately, we have a covering index for the task (*idx_unc_od_oid_i_cid_eid*), with the key columns being (*orderdate, orderid*), and included non-key columns are (*custid, empid*). I'll elaborate on included non-key columns later in the chapter. The important point for our discussion is that the segment operator organizes the data by groups of *orderdate* values and emits the data, a group at a time, where the last row in each group is the maximum *orderid* in the group; because *orderid* is the second key column right after *orderdate.* Therefore, there's no need for the plan to sort the data; rather, the plan just collects it with an ordered scan from the covering index, which is already sorted by *orderdate* and *orderid.* The *Top* operator has a simple task of just collecting the last row (TOP 1 descending), which is the row of interest for the group. The number of rows reported by the *Top* operator is 1491, which is the number of unique groups (*orderdate* values), each of which got a single row from the operator. Because our nonclustered index covers the query by including in its leaf level all other columns that are mentioned in the query (*custid, empid*), there's no need to look up the data rows; the query is satisfied by the index data alone. Here are the performance measures I got for this query:

- Logical reads: 4720
- Physical reads: 3
- Read-ahead reads: 4695
- CPU time: 781 ms
- Elapsed time: 2128 ms
- Estimated subtree cost: 4.68047

The number of logical reads that you see is similar to the number of pages that the leaf level of the index holds.

Nonclustered Index Seek + Ordered Partial Scan + Lookups

The access method *nonclustered index seek + ordered partial scan + lookups* is typically used for small-range queries (including a point query) using a nonclustered index scan that doesn't cover the query. To demonstrate this access method, I will use the following query:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid BETWEEN 101 AND 120;
```

There's no covering index because the first key column is the filtered column *orderid*, but we do have a noncovering one—the *PK_Orders* index. If the query is *selective* enough, the optimizer would use the index. Selectivity is defined as the percentage of the number of rows returned by the query out of the total number of rows in the table. The term *high selectivity* refers to a small percentage, while *low selectivity* refers to a large percentage. Our access method first performs a seek within the index to find the first key in the sought range (*orderid = 101*). The second part of the access method is an ordered partial scan in the leaf level from the first key in the range until the last (*orderid = 101*). The third and last part involves lookups of the corresponding data row for each key. Note that the third part doesn't have to wait for the second part to finish. For each key that is found in the range, SQL Server can already apply a lookup. Remember that a lookup in a heap translates to a single page read, while a lookup in a clustered table translates to as many reads as the number of levels in the clustered index (3 in our case).

It is vital for making performance estimations to understand that with this access method the last part involving the lookups typically incurs most of the query's cost; this is because it involves most of the I/O activity. Remember, the lookup translates to a whole page read or one whole seek within the clustered index per sought row, and the lookups are always random I/O (as opposed to sequential ones).

To estimate the I/O cost of such a query, you can typically focus on the cost of the lookups. If you want to make more accurate estimations, also taking into consideration the seek within the index and the ordered partial scan, feel free to do so; but these parts will be negligible as the range grows larger. The I/O cost of a seek operation is 3 reads in our case (number of levels in the index). The I/O cost of the ordered partial scan depends on the number of rows in the range (20 in our case) and the number of rows that fit in an index page (more than 600 in our case). For our query, there's actually no additional read involved for the partial scan; that's because all the keys in the range we are after reside in the leaf page that the seek reached, or they might span an additional page if the first key appears close to the end of the page. The I/O cost of the lookup operations will be the number of rows in the range (20 in our case), multiplied by one if the table is a heap, or multiplied by the number of levels in the clustered index (3 in our case) if the table is clustered. So you should expect around 23 logical reads in total if you run the query against a heap, and around 63 logical reads if you run it against a clustered table. Remember that the non-leaf levels of the clustered index typically reside in cache because of all the lookup operations going through it; so you shouldn't concern yourself too much over the seemingly higher cost of the query in the clustered table scenario.

Figure 3-32 has an illustration of the access method over a heap, and Figure 3-33 shows the execution plan for the query.

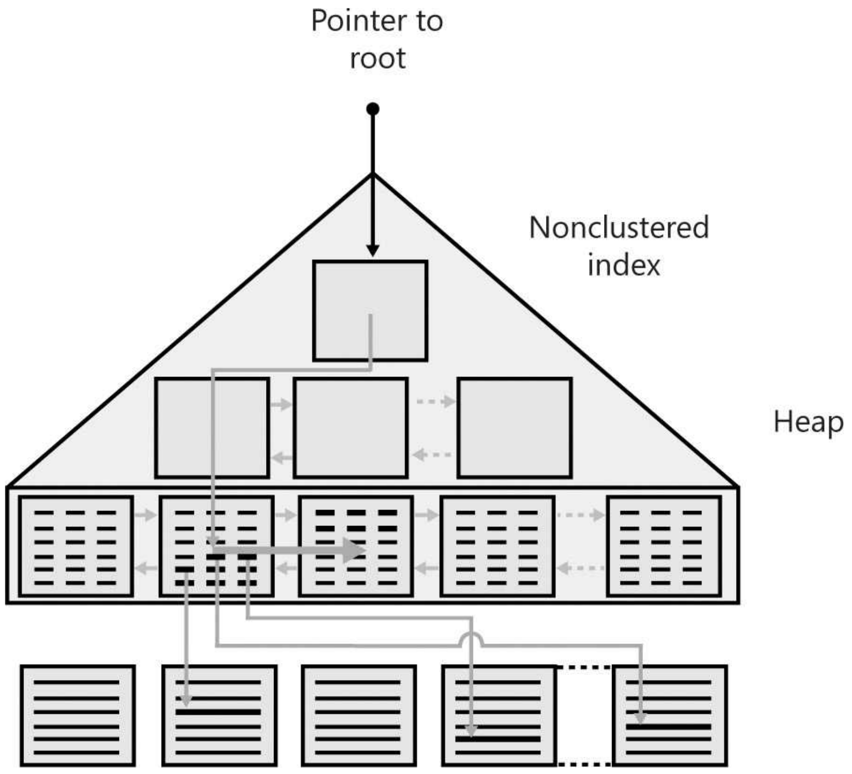
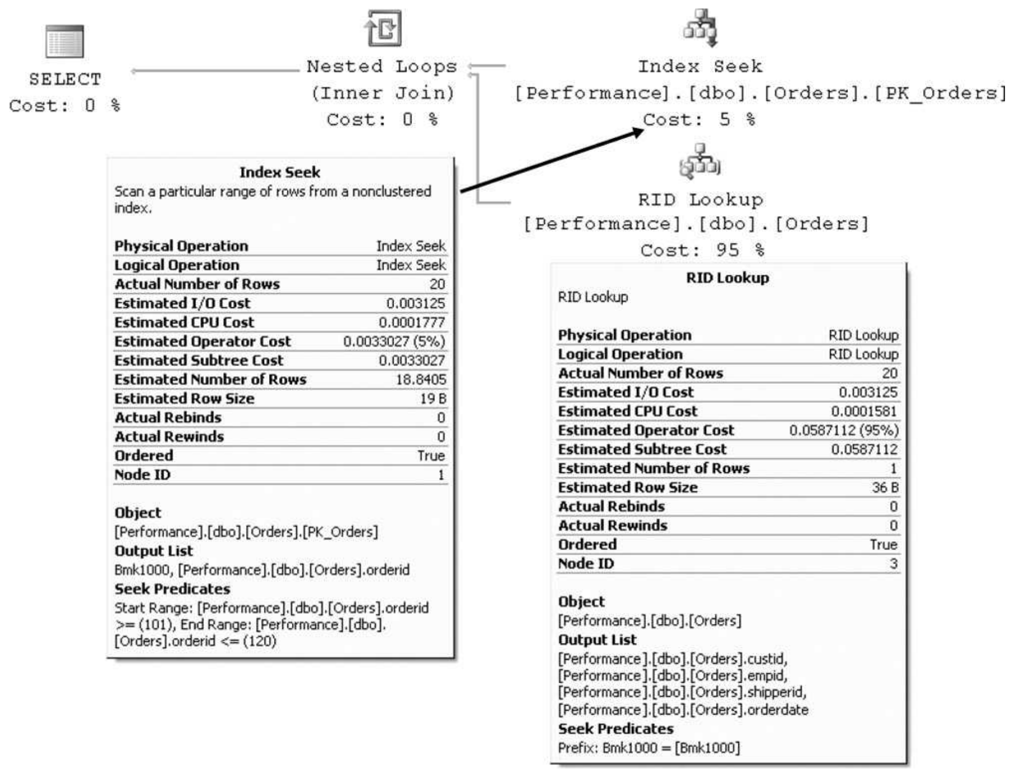


Figure 3-32: Nonclustered index seek + ordered partial scan + lookups against a heap



Index Seek

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	20
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001777
Estimated Operator Cost	0.0033027 (5%)
Estimated Subtree Cost	0.0033027
Estimated Number of Rows	18.8405
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	1

Object
[Performance].[dbo].[Orders].[PK_Orders]
Output List
Bmk1000, [Performance].[dbo].[Orders].orderid
Seek Predicates
Start Range: [Performance].[dbo].[Orders].orderid >= (101), End Range: [Performance].[dbo].[Orders].orderid <= (120)

RID Lookup

RID Lookup

Physical Operation	RID Lookup
Logical Operation	RID Lookup
Actual Number of Rows	20
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Operator Cost	0.0587112 (95%)
Estimated Subtree Cost	0.0587112
Estimated Number of Rows	1
Estimated Row Size	36 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3

Object
[Performance].[dbo].[Orders]
Output List
[Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].empid, [Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].orderdate
Seek Predicates
Prefix: Bmk1000 = [Bmk1000]

Figure 3-33: Nonclustered index seek + ordered partial scan + lookups against a heap (execution plan)

Note that in the execution plan you won't explicitly see the partial scan part of the access method; rather, it's hidden in the *Index Scan* operator. You can deduce it from the fact that the information box for the operator shows *True* in the *Ordered* measure.

Here are the performance measures I got for the query:

- Logical reads: 23
- Physical reads: 22
- CPU time: 0 ms
- Elapsed time: 133 ms
- Estimated subtree cost: 0.0620926

Figure 3-34 has an illustration of the access method over a clustered table, and Figure 3-35 shows the execution plan for the query.

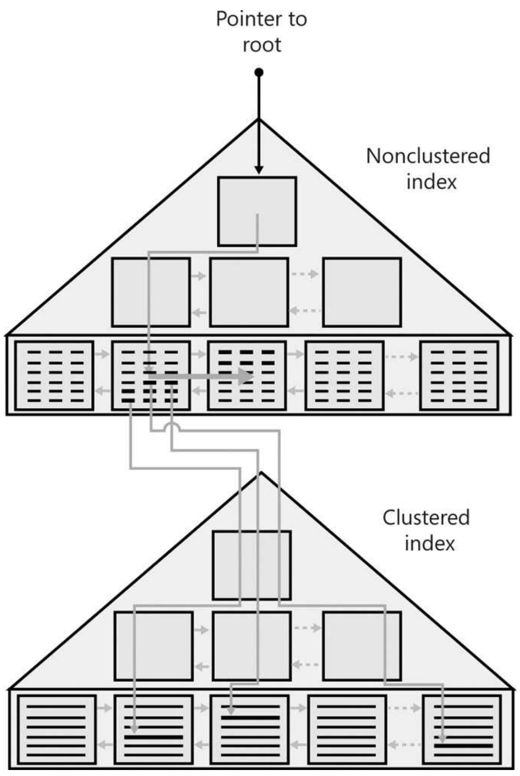
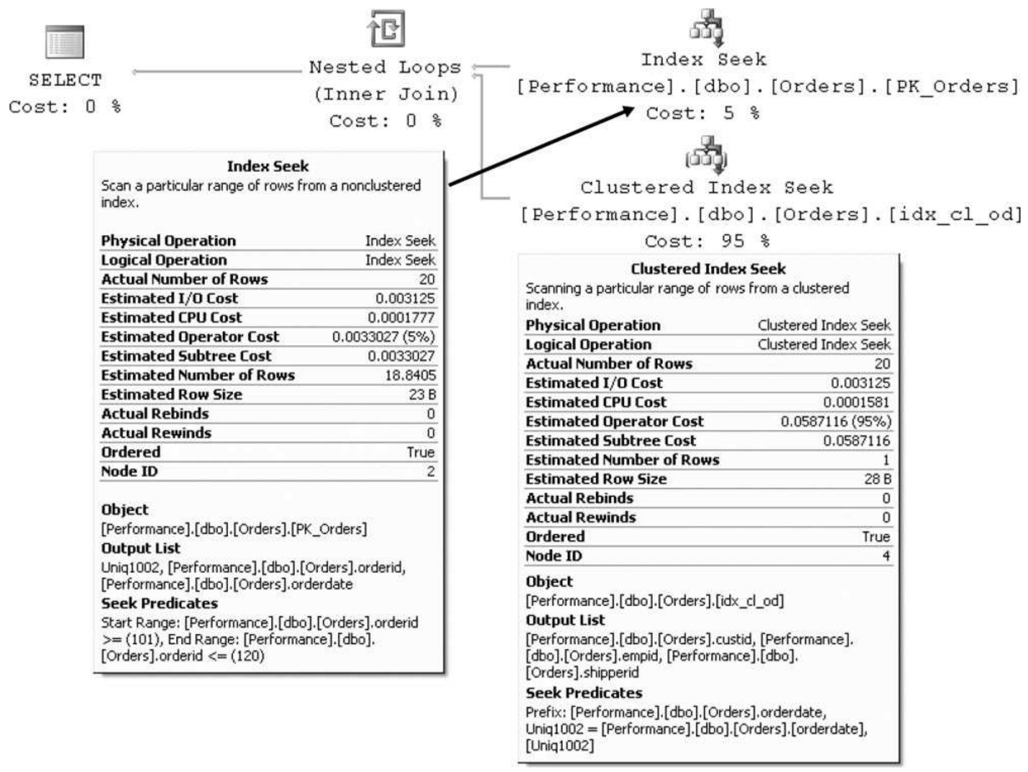


Figure 3-34: Nonclustered index seek + ordered partial scan + lookups against a clustered tale


SELECT
Cost: 0 %

Nested Loops
(Inner Join)
Cost: 0 %

Index Seek
[Performance].[dbo].[Orders].[PK_Orders]
Cost: 5 %

Clustered Index Seek
[Performance].[dbo].[Orders].[idx_cl_od]
Cost: 95 %

Index Seek

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	20
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001777
Estimated Operator Cost	0.0033027 (5%)
Estimated Subtree Cost	0.0033027
Estimated Number of Rows	18.8405
Estimated Row Size	23 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	2

Object
[Performance].[dbo].[Orders].[PK_Orders]
Output List
Uniq1002, [Performance].[dbo].[Orders].orderid, [Performance].[dbo].[Orders].orderdate
Seek Predicates
Start Range: [Performance].[dbo].[Orders].orderid >= (101), End Range: [Performance].[dbo].[Orders].orderid <= (120)

Clustered Index Seek

Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	20
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Operator Cost	0.0587116 (95%)
Estimated Subtree Cost	0.0587116
Estimated Number of Rows	1
Estimated Row Size	28 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	4

Object
[Performance].[dbo].[Orders].[idx_cl_od]
Output List
[Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].empid, [Performance].[dbo].[Orders].shipperid
Seek Predicates
Prefix: [Performance].[dbo].[Orders].orderdate, Uniq1002 = [Performance].[dbo].[Orders].[orderdate], [Uniq1002]

Figure 3-35: Nonclustered index seek + ordered partial scan + lookups against a clustered tale (execution plan)

And here are the performance measures I got for the query in this case:

- Logical reads: 63
- Physical reads: 6
- CPU time: 0 ms
- Elapsed time: 136 ms
- Estimated subtree cost: 0.0620931

Interestingly, graphical execution plans in SQL Server 2000 did not clearly distinguish between an RID lookup and a clustering key lookup, the latter in fact being a seek within the clustered index. In SQL Server 2000, both were just called Bookmark Lookup. As you can see in the plans shown in Figures 3-33 and 3-35, graphical plans in SQL Server 2005 depict the difference more accurately.

This access method is efficient only when the query is very selective (a point query or a small range). Feel free to play with the range in the filter, increasing it gradually, and see how dramatically the cost increases as the range grows larger. That will happen up to the point at which the optimizer figures that it would simply be more efficient to apply a table scan rather than using the index. I'll demonstrate such an exercise later in the chapter in the "Index Optimization Scale" section.

Remember that ordered operations, like the ordered partial scan part of this access method, can take place both in a forward or backwards manner. In our query's case, it was forward, but you can explore cases like a filter on *orderid <=100*, and see an ordered backwards partial scan. An indication of whether the scan was ordered forward or backwards would not show up in the operator's yellow information box; rather, it would show up in the *Scan Direction* measure in the operator's Properties dialog box.

Unordered Nonclustered Index Scan + Lookups

The optimizer typically uses the *unordered nonclustered index scan + lookups* access method when the following conditions are in place:

- The query is selective enough

- The optimal index for a query does not cover it
- The index doesn't maintain the sought keys in order

For example, such is the case when you filter a column that is not the first key column in the index. The access method will involve an unordered full scan of the leaf level of the index, followed by a series of lookups. As I mentioned, the query must be selective enough to justify this access method; otherwise, with too many lookups it will be more expensive than simply scanning the whole table. To figure out the selectivity of the query, SQL Server will need statistics on the filtered column (a histogram with the distribution of values). If such statistics do not exist, SQL Server will create them.

For example, the following query will use such an access method against the index *idx_nc_sid_od_cid*, created on the key columns (*shipperid, orderdate, custid*), where *custid* is not the first key column in the list:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE custid ='C0000000001';
```

Figure 3-36 illustrates the access method over a heap, and Figure 3-37 shows the execution plan for the query.

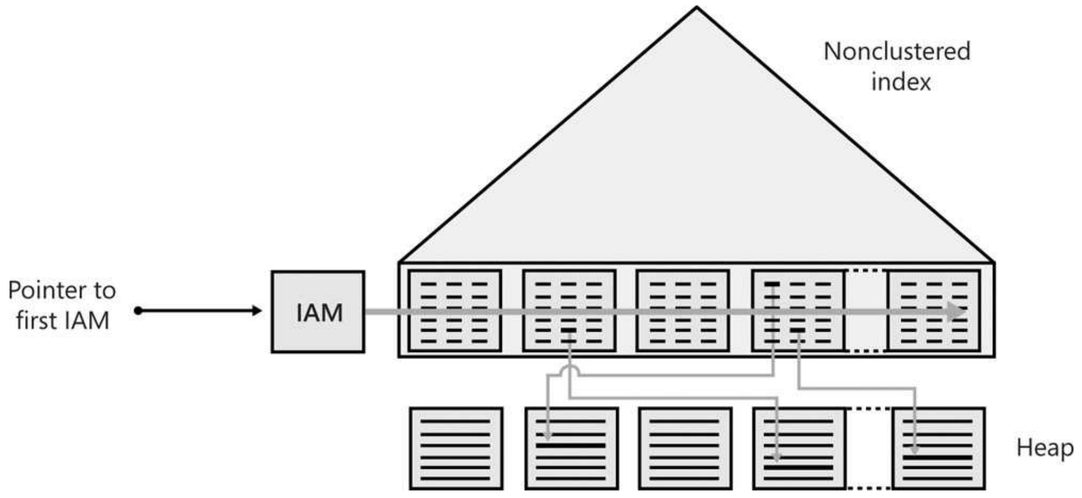
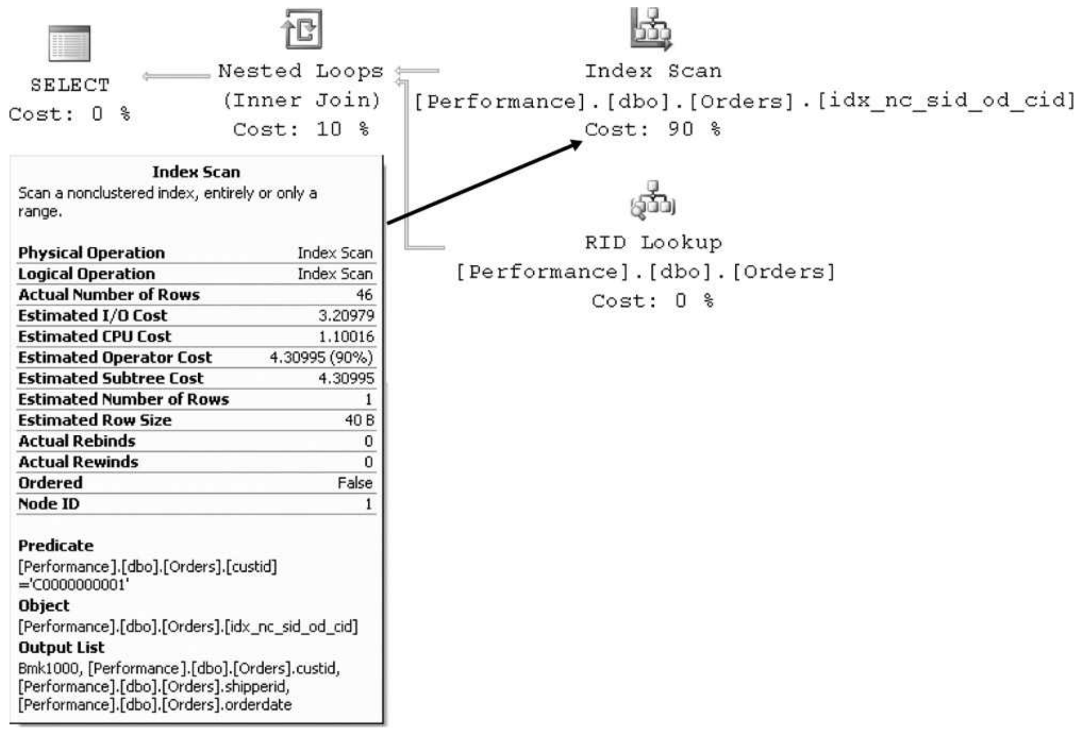


Figure 3-36: Unordered nonclustered index scan + lookups against a heap



Index Scan

Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Number of Rows	46
Estimated I/O Cost	3.20979
Estimated CPU Cost	1.10016
Estimated Operator Cost	4.30995 (90%)
Estimated Subtree Cost	4.30995
Estimated Number of Rows	1
Estimated Row Size	40 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

Predicate
[Performance].[dbo].[Orders].[custid] ='C0000000001'
Object
[Performance].[dbo].[Orders].[idx_nc_sid_od_cid]
Output List
Bmk1000, [Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].orderdate

Figure 3-37: Unordered nonclustered index scan + lookups against a heap (execution plan)

The I/O cost of this query involves the cost of the unordered scan of the leaf of the index (sequential I/O using IAM pages) plus the cost of the lookups (random I/O). The scan will cost as many page reads as the number of pages in the leaf of the index. As described earlier, the cost of the lookups is the number of qualifying rows multiplied by 1 in a heap, and multiplied by the number of levels in the clustered index (3 in our case) if the table is clustered. Here are the measures I got for this query against a heap:

- Logical reads: 4400
- Physical reads: 47
- Read-ahead reads: 4345
- CPU time: 1281 ms
- Elapsed time: 2287 ms
- Estimated subtree cost: 4.479324

Figure 3-38 illustrates the access method over a clustered table, and Figure 3-39 shows the execution plan for the query.

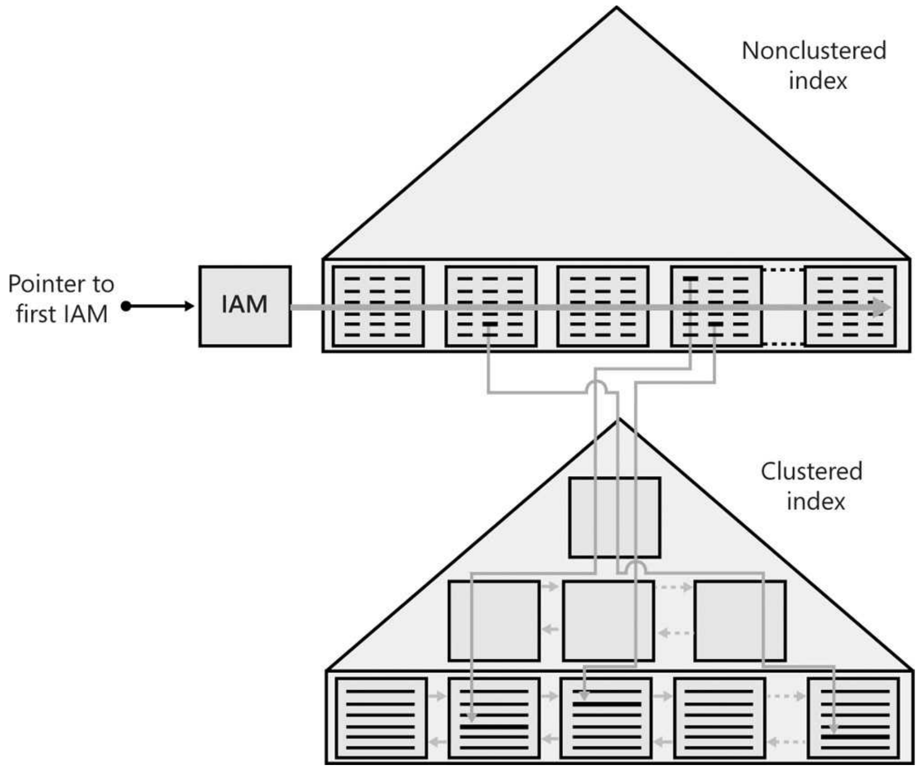
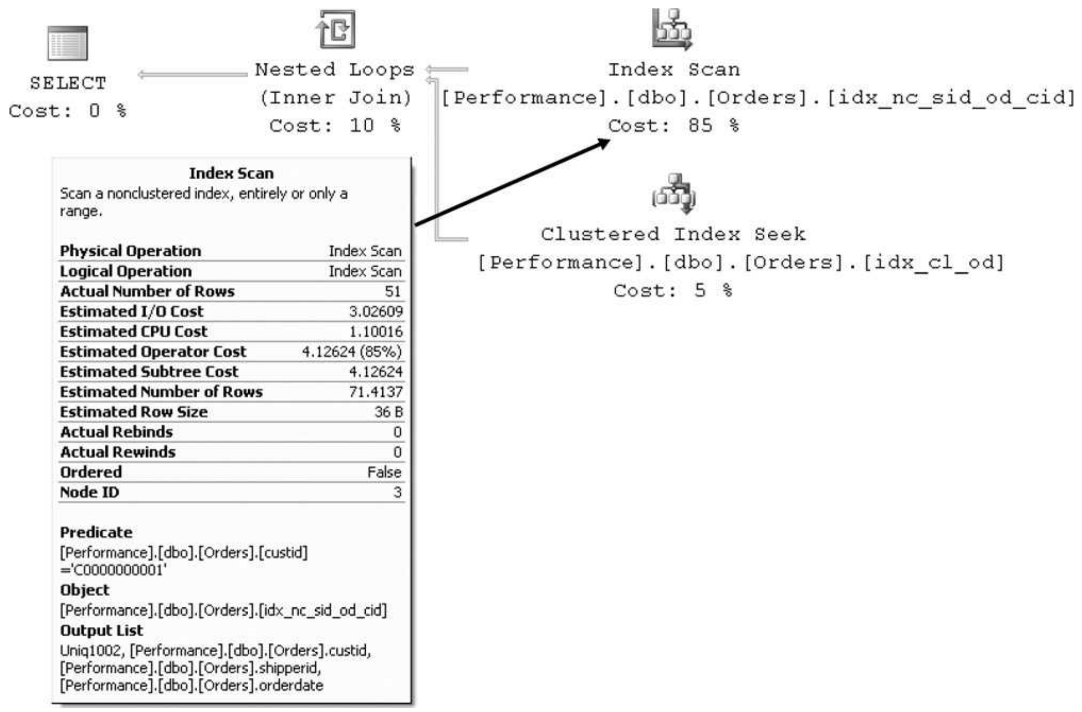


Figure 3-38: Unordered nonclustered index scan + lookups against a clustered table



Index Scan

Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Number of Rows	51
Estimated I/O Cost	3.02609
Estimated CPU Cost	1.10016
Estimated Operator Cost	4.12624 (85%)
Estimated Subtree Cost	4.12624
Estimated Number of Rows	71.4137
Estimated Row Size	36 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	3

Predicate
[Performance].[dbo].[Orders].[custid] ='C0000000001'
Object
[Performance].[dbo].[Orders].[idx_nc_sid_od_cid]
Output List
Uniq1002, [Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].orderdate

Figure 3-39: Unordered nonclustered index scan + lookups against a clustered table (execution plan 1)

Here are the measures I got for this query against a clustered table:

- Logical reads: 4252
- Physical reads: 89
- Read-ahead reads: 4090
- CPU time: 1031 ms
- Elapsed time: 3148 ms
- Estimated subtree cost: 4.60953

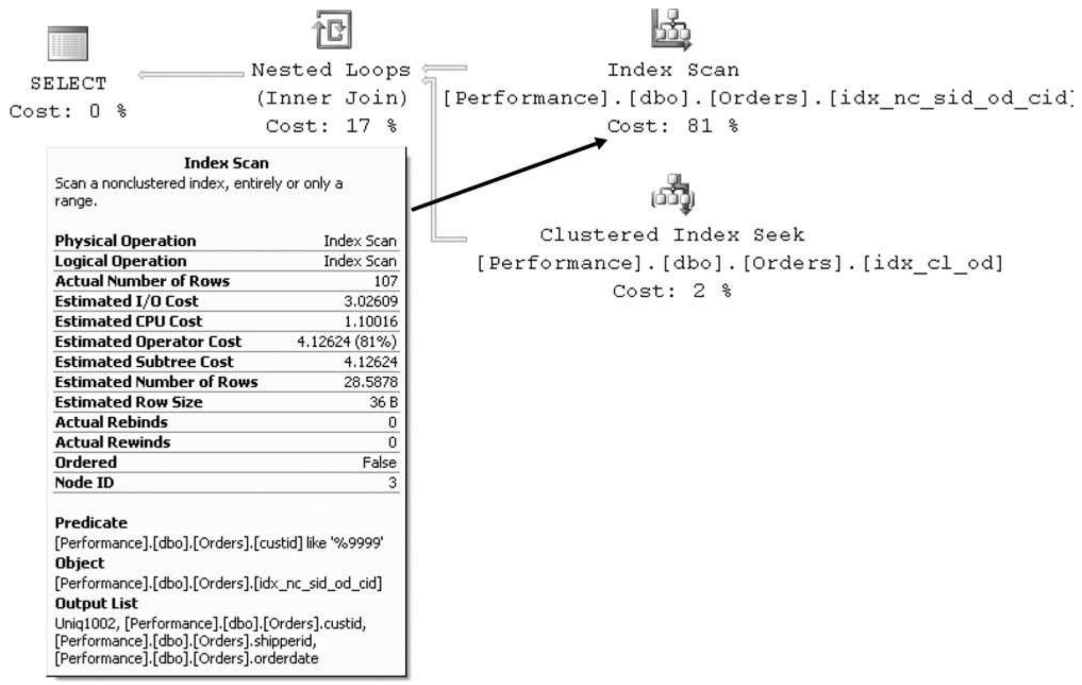
Remember that SQL Server will need statistics on the <emphasis>custid</emphasis> column to determine the selectivity of the query. The following query will tell you which statistics SQL Server created automatically on the Orders table:

```
SELECT name
FROM sys.stats
WHERE object_id = OBJECT_ID('dbo.Orders')
  AND auto_created = 1;
```

You should get statistics with a name similar to *_WA_Sys_00000002_31B762FC*, which SQL Server created automatically for this purpose.

SQL Server 2005 introduces new optimization capabilities based on cardinality information that it maintains internally on substrings within string columns. Now it can estimate the selectivity of a query when you apply pattern-matching filters with the LIKE predicate even when the pattern starts with a wildcard. This capability was not available in earlier versions of SQL Server.

To demonstrate this capability, SQL Server will be able to estimate the selectivity of the following query, which produces the plan shown in Figure 3-40 by using the access method, which is the focus of this section's discussion:



Index Scan

Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Number of Rows	107
Estimated I/O Cost	3.02609
Estimated CPU Cost	1.10016
Estimated Operator Cost	4.12624 (81%)
Estimated Subtree Cost	4.12624
Estimated Number of Rows	28.5878
Estimated Row Size	36 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	3

Predicate
[Performance].[dbo].[Orders].[custid] like '%9999'
Object
[Performance].[dbo].[Orders].[idx_nc_sid_od_cid]
Output List
Uniq1002, [Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].orderdate

Figure 3-40: Unordered nonclustered index scan + lookups against a clustered table (execution plan 2)

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE custid LIKE '%9999';
```

Here are the performance measures that I got for this query:

- Logical reads: 4685
- Physical reads: 1
- Read-ahead reads: 3727
- CPU time: 3795 ms
- Elapsed time: 4824 ms
- Estimated subtree cost: 5.09967

Clustered Index Seek + Ordered Partial Scan

The optimizer typically uses the access method *clustered index seek + ordered partial scan* for range queries where you filter based on the first key columns of the clustered index. This access method first performs a seek operation to the first key in the range; then it applies an ordered partial scan at the leaf level from the first key in the range until the last. The main benefit of this method is that there are no lookups involved. Remember that lookups are very expensive with large ranges. The performance ratio between this access method—which doesn't involve lookups—and one that uses a nonclustered index and lookups becomes larger and larger as the range grows.

The following query, which looks for all orders placed on a given *orderdate*, uses the access method, which is the focus of this discussion:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderdate = '20060212';
```

Note that even though the filter uses an equality operator, it is in essence a range query because there are multiple qualifying rows. Either way, a point query can be considered a special case of a range query. The I/O cost of this access method will involve the cost of the seek operation (3 random reads in our case) and the cost of the ordered partial scan within the leaf (in our case, 18 page reads). In total, you get 21 logical reads. Note that the ordered scan typically incurs the bulk of the cost of the query because it involves most of the I/O. Remember that with ordered index scans, index fragmentation plays a crucial role. When fragmentation is at minimum (as in our case), physical reads will be close to sequential. However, as the fragmentation level grows higher, the disk arm will have to move frantically to and fro, degrading the performance of the scan.

Figure 3-41 illustrates the access method, and Figure 3-42 shows the execution plan for the query.

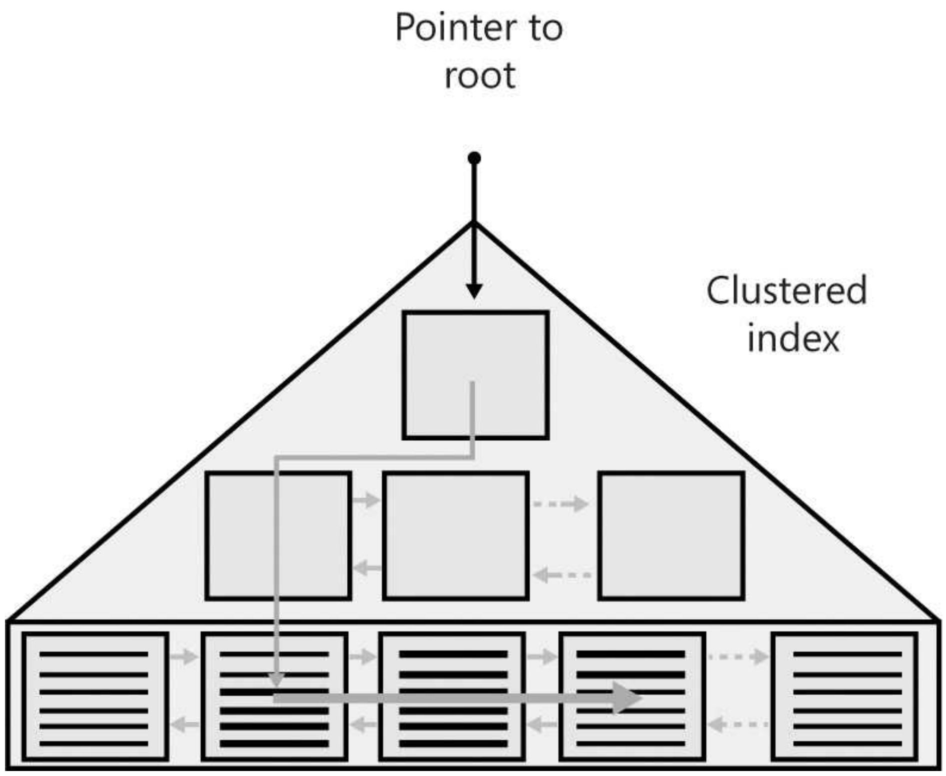
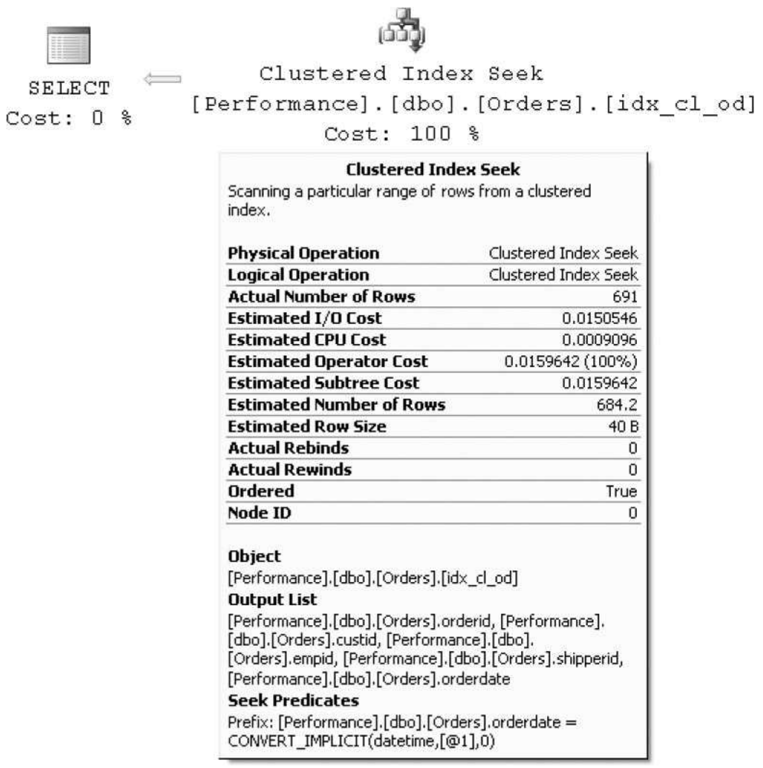


Figure 3-41: Clustered index seek + ordered partial scan



Clustered Index Seek
Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	691
Estimated I/O Cost	0.0150546
Estimated CPU Cost	0.0009096
Estimated Operator Cost	0.0159642 (100%)
Estimated Subtree Cost	0.0159642
Estimated Number of Rows	684.2
Estimated Row Size	40 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0

Object
[Performance].[dbo].[Orders].[idx_cl_od]
Output List
[Performance].[dbo].[Orders].orderid, [Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].empid, [Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].orderdate
Seek Predicates
Prefix: [Performance].[dbo].[Orders].orderdate = CONVERT_IMPLICIT(datetime,[@1],0)

Figure 3-42: Clustered index seek + ordered partial scan (execution plan)

Here are the performance measures I got for this query:

- Logical reads: 21
- Physical reads: 0
- Read-ahead reads: 18
- CPU time: 0 ms
- Elapsed time: 73 ms
- Estimated subtree cost: 0.0159642

Note that this plan is trivial for the optimizer to generate. That is, the plan is not dependent on the selectivity of the query. Rather, it will always be used regardless of the size of the sought range. Unless, of course, you have an even better index for the query to begin with.

Covering Nonclustered Index Seek + Ordered Partial Scan

The access method *covering nonclustered index seek* + *ordered partial scan* is almost identical to the previously described access method, with the only difference being that the former uses a covering nonclustered index instead of the clustered index. To use this method, of course the filtered columns must be the first key columns in the index. The benefit of this access method over the previous one lies in the fact that a nonclustered index leaf page naturally can fit more rows than a clustered index one; therefore, the bulk cost of the plan, which is the partial scan cost of the leaf, is lower. The cost is lower because fewer pages need to be scanned for the same size of the range. Of course, here as well, index fragmentation plays an important performance role because the partial scan is ordered.

As an example, the following query looking for a range of *orderdate* values for a given *shipperid* uses this access method against the covering index *idx_nc_sid_od_cid*, created on (*shipperid, orderdate, custid*):

```
SELECT shipperid, orderdate, custid
FROM dbo.Orders
WHERE shipperid ='C'
  AND orderdate >= '20060101'
  AND orderdate < '20070101';
```

Note To have the partial scan read the minimum required pages, the first index key columns must be *shipperid, orderdate*, in that order. If you swap their order, the partial scan will end up also scanning rows that meet the date range also for other shippers, requiring more I/O.

Figure 3-43 illustrates the access method, and Figure 3-44 shows the execution plan for the query.

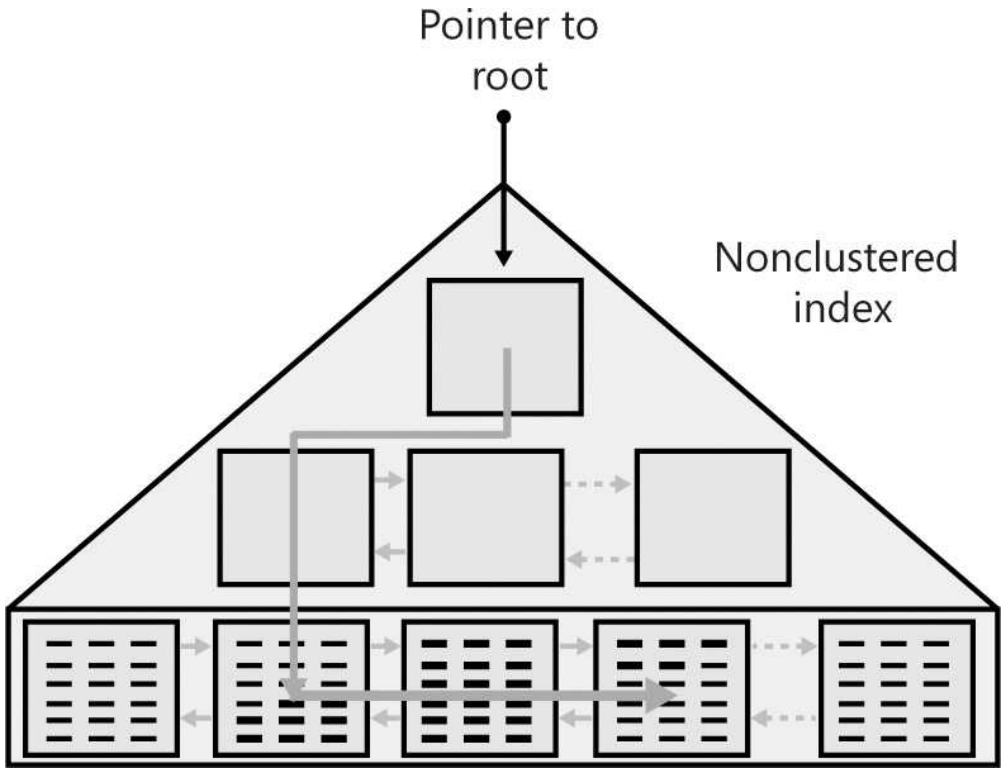
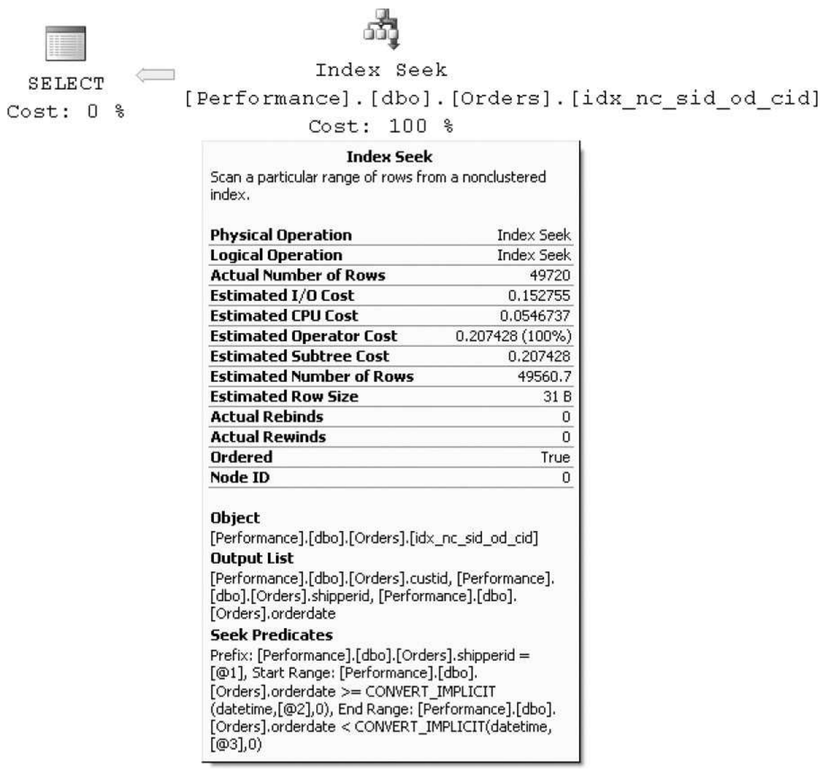


Figure 3-43: Covering nonclustered index seek + ordered partial scan



Index Seek

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	49720
Estimated I/O Cost	0.152755
Estimated CPU Cost	0.0546737
Estimated Operator Cost	0.207428 (100%)
Estimated Subtree Cost	0.207428
Estimated Number of Rows	49560.7
Estimated Row Size	31 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0

Object
[Performance].[dbo].[Orders].[idx_nc_sid_od_cid]
Output List
[Performance].[dbo].[Orders].custid, [Performance].[dbo].[Orders].shipperid, [Performance].[dbo].[Orders].orderdate
Seek Predicates
Prefix: [Performance].[dbo].[Orders].shipperid = [@1], Start Range: [Performance].[dbo].[Orders].orderdate >= CONVERT_IMPLICIT(datetime,[@2],0), End Range: [Performance].[dbo].[Orders].orderdate < CONVERT_IMPLICIT(datetime,[@3],0)

Figure 3-44: Covering nonclustered index seek + ordered partial scan (execution plan)

Here are the performance measures I got for this query:

- Logical reads: 208
- CPU time: 30 ms
- Elapsed time: 954 ms

- Estimated subtree cost: 0.207428

Note that this plan is also a trivial plan that is not based on the query's selectivity.

Remember, the main benefit of this access method is that there are no lookups involved because the index covers the query. Also, you read fewer pages than in a similar access method against a clustered index.

Also note that when you create covering indexes, the index columns serve two different functions. Columns that you filter or sort by are required as key columns that will be maintained in all levels of the balanced tree, and they will also determine the sort order at the leaf. Other index columns might be required only for covering purposes. If you include all index columns in the index's key column list, bear in mind that this has a cost. SQL Server needs to keep the tree balanced, and it will have to apply physical movement of data and adjustments in the tree when you modify key column values in the table. That's just a waste with columns that are required only for covering purposes and not for filtering or sorting.

To tackle this need, SQL Server 2005 introduces the concept of *included non-key columns* in the index. When you create an index, you separately specify which columns will make the key list and which will be included just for covering purposes—only at the leaf level of the index.

As an example, our last query relied only on *shipperid* and *orderdate* for filtering and sorting purposes, while it relied on *custid* only for covering purposes. To benefit from the new feature in SQL Server 2005, drop the index and create a new one, specifying *custid* in the INCLUDE clause like so:

```
DROP INDEX dbo.Orders.idx_nc_sid_od_cid;

CREATE NONCLUSTERED INDEX idx_nc_sid_od_i_cid
  ON dbo.Orders(shipperid, orderdate)
  INCLUDE(custid);
```

Note that the key list is limited to 16 columns and 900 bytes. This is true in both SQL Server 2000 and SQL Server 2005. An added bonus with included non-key columns is that they are not bound by the same limitations. In fact, they can even include large objects such as variable-length columns defined with the MAX specifier and XML columns.

Index Intersection

So far, I mainly focused on the performance benefit you get from indexes when reading data. Keep in mind, though, that indexes incur a cost when you modify data. Any change of data (deletes, inserts, updates) must be reflected in the indexes that hold a copy of that data, and it might cause page splits and adjustments in the balanced trees, which can be very expensive. Therefore, you cannot freely create as many indexes as you like, especially in systems that involve intensive modifications like OLTP environments. You want to prioritize and pick the more important indexes. This is especially a problem with covering indexes because different queries can benefit from completely different covering indexes, and you might end up with a very large number of indexes that your queries could benefit from.

Fortunately, the problem is somewhat reduced because the optimizer supports a technique called *index intersection*, where it intersects data obtained from two indexes and, if required, then intersects the result with data obtained from another index, and so on. As an example, the optimizer will use index intersection for the following query, producing the plan shown in Figure 3-45:

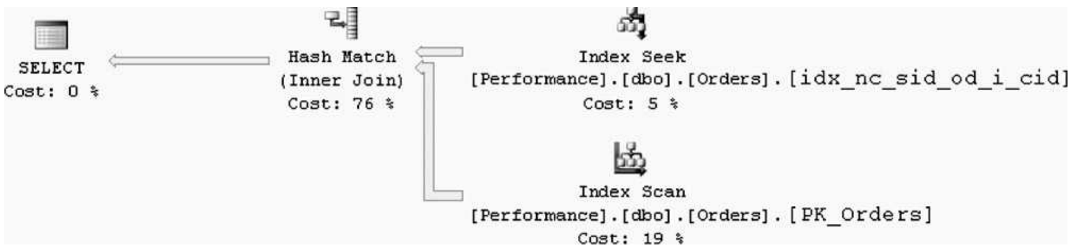

SELECT
Cost: 0 %
Hash Match
(Inner Join)
Cost: 76 %
Index Seek
[Performance].[dbo].[Orders].[idx_nc_sid_od_i_cid]
Cost: 5 %
Index Scan
[Performance].[dbo].[Orders].[PK_Orders]
Cost: 19 %

Figure 3-45: Execution plan with index intersection

```
SELECT orderid, custid
FROM dbo.Orders
WHERE shipperid = 'A';
```

I will elaborate on join operators in Chapter 5. The optimal index here would be one where *shipperid* is defined as the key column, and *orderid* and *custid* are defined as included non-key columns, but there's no such index on the table. Rather, the index *idx_nc_sid_od_i_cid* defines the *shipperid* as the key column and also contains the *custid* column, and the index *PK_Orders* contains the *orderid* column. The optimizer used the access method nonclustered index seek + ordered partial scan to obtain the relevant data from *idx_nc_sid_od_i_cid*, and it used an unordered nonclustered index scan to obtain the relevant data from *PK_Orders.* It then intersected the two sets based on the row locator values; naturally, row locator values pointing to the same rows will be matched. You can think of index intersection as an internal join based on a match in row locator values.

Here are the performance measures that I got for this query:

- Scan count: 2
- Logical reads: 3671
- Physical reads: 33
- Read-ahead reads: 2347
- CPU time: 1161 ms
- Elapsed time: 5202 ms
- Estimated subtree cost: 16.7449

Indexed Views

This section briefly describes and demonstrates the concept of *indexed views* for the sake of completeness. I won't conduct a lengthy discussion on the subject here. I'll provide a bit more details in *Inside T-SQL Programming* and point you to more resources at the end of the chapter.

As of SQL Server 2000, you can create indexes on views—not just on tables. Normally, a view is a virtual object, and a query against it ultimately queries the underlying tables. However, when you create a clustered index on a view, you materialize all of the view's contents within the clustered index on disk. After creating a clustered index, you can also create multiple nonclustered indexes on the view as well. The data in the indexes on the view will be kept in sync with the changes in the underlying tables as with any other index.

Indexed views are mainly beneficial in reducing I/O costs and expensive processing of data. Such costs are especially apparent in aggregation queries that scan large volumes of data and produce small result sets, and in expensive join queries.

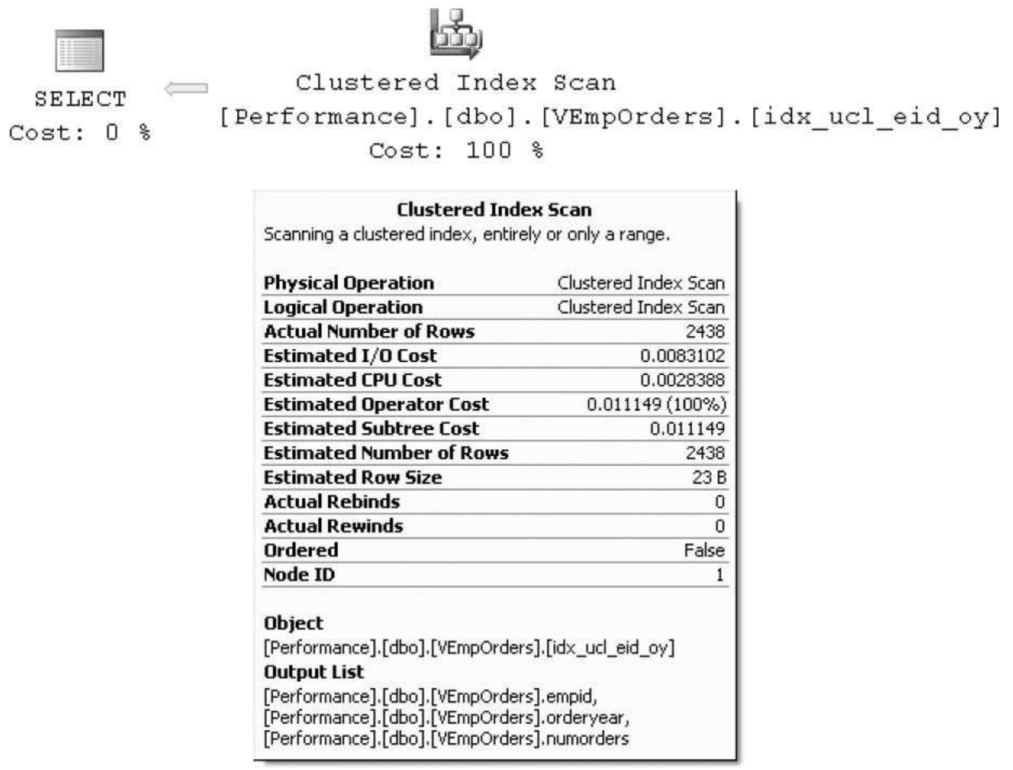
As an example, the following code creates an indexed view that is designed to tune aggregate queries that group orders by *empid* and *YEAR(orderdate)*, returning the count of orders for each group:

```
IF OBJECT_ID('dbo.VEmpOrders') IS NOT NULL
  DROP VIEW dbo.VEmpOrders;
GO
CREATE VIEW dbo.VEmpOrders
  WITH SCHEMABINDING
AS

SELECT empid, YEAR(orderdate) AS orderyear, COUNT_BIG(*) AS numorders
FROM dbo.Orders
GROUP BY empid, YEAR(orderdate);
GO

CREATE UNIQUE CLUSTERED INDEX idx_ucl_eid_oy
  ON dbo.VEmpOrders(empid, orderyear);
```

Query the view, and you will get the execution plan shown in Figure 3-46, showing that the clustered index on the view was scanned:



Clustered Index Scan
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	2438
Estimated I/O Cost	0.0083102
Estimated CPU Cost	0.0028388
Estimated Operator Cost	0.011149 (100%)
Estimated Subtree Cost	0.011149
Estimated Number of Rows	2438
Estimated Row Size	23 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

Object
[Performance].[dbo].[VEmpOrders].[idx_ucl_eid_oy]
Output List
[Performance].[dbo].[VEmpOrders].empid,
[Performance].[dbo].[VEmpOrders].orderyear,
[Performance].[dbo].[VEmpOrders].numorders

Figure 3-46: Execution plan for query against indexed view

```
SELECT empid, orderyear, num orders
FROM dbo.VEmpOrders;
```

The view contains a very small number of rows (around a couple of thousands) compared to the number of rows in the table (a million). The leaf of the index contains only about 10 pages. Hence, the I/O cost of the plan would be about 10 page reads.

Here are the performance measures I got for this query:

- Logical reads: 10
- CPU time: 0 ms
- Elapsed time: 78 ms
- Estimated subtree cost: 0.011149

Interestingly, if you work with an Enterprise (or Developer) edition of SQL Server, the optimizer will consider using indexes on the view even when querying the underlying tables directly. For example, the following query produces a similar plan to the one shown in Figure 3-46, with the same query cost:

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT_BIG(*) AS numorders
FROM dbo.Orders
GROUP BY empid, YEAR(orderdate);
```

If you're not working with an Enterprise edition, you have to query the view directly, and also specify that you do not want the optimizer to expand its optimization choices beyond the scope of the view. You do so by specifying the NOEXPAND table hint: *FROM <view_name> WITH (NOEXPAND).*

SQL Server 2005 enhances the circumstances in which indexed views are used, supporting subintervals, logically equivalent filter expressions, and more. As I mentioned, I'll point you to resources that describe these in detail.

Index Optimization Scale

Recall the earlier discussion about the tuning methodology. When you perform index tuning, you do so with respect to the query patterns that incur the highest cumulative costs in the system. For a given query pattern, you can build an index optimization scale that would help you make the right design choices. I will demonstrate this process through an example.

To follow the demonstrations, before you continue, drop the view created earlier and all the indexes on the Orders table, except for the clustered index. Alternatively, you can rerun the code in Listing 3-1, after commenting or removing all index and primary key creation statements on Orders, keeping only the clustered index.

In our example, suppose that you need to tune the following query pattern:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= value;
```

Remember that the efficiency of some access methods depends on the selectivity of the query, while the efficiency of others doesn't. For access methods that depend on selectivity, assume that the query pattern is typically fairly selective (around 0.1 percent selectivity, or around 1000 qualifying rows). Use the following query in your tuning process when aiming at such selectivity:

```
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= 999001;
```

I'll progress in the index optimization scale from the worst-case scenario to the best, using this query as a reference, but I'll also describe what would happen when the selectivity of the query changes.

Table Scan (Unordered Clustered Index Scan)

The worst-case scenario for our query pattern with fairly high selectivity is when you have no good index. You will get the execution plan shown in Figure 3-47, using a table scan (unordered clustered index scan).

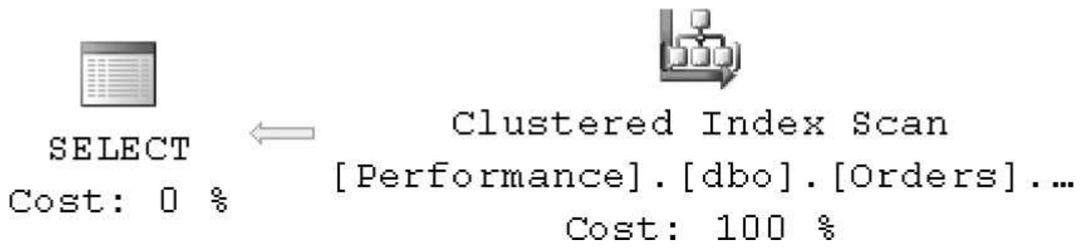


Figure 3-47: Execution plan with table scan (unordered clustered index scan)

Even though you're after a fairly small number of rows (1000 in our case), the whole table is scanned. I got the following performance measures for this query:

- Logical reads: 25080
- CPU time: 1472 ms
- Elapsed time: 16399
- Estimated subtree cost: 19.6211

This plan is trivial and not dependent on selectivity—that is, you would get the same plan regardless of the selectivity of the query.

Unordered Covering Nonclustered Index Scan

The next step in the optimization scale would be to create a covering nonclustered index where the filtered column (*orderid*) is not the first index column:

```
CREATE NONCLUSTERED INDEX idx_nc_od_i_oid_cid_eid_sid
  ON dbo.Orders(orderdate)
  INCLUDE(orderid, custid, empid, shipperid);
```

This index would yield an access method that uses a full unordered scan of the leaf of the index as shown in Figure 3-48:

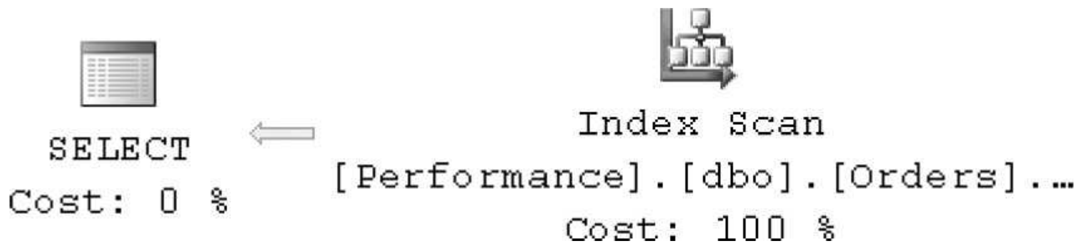


Figure 3-48: Execution plan with unordered covering nonclustered index scan

The row size in the covering index is about a fifth of the size of a full data row, and this would be reflected in the query's cost and run time. Here are the performance measures I got for this query:

- Logical reads: 5095
- CPU time: 170 ms
- Elapsed time: 1128 ms
- Estimated subtree cost: 4.86328

As with the previous plan, this plan is also trivial and not dependent on selectivity.

Note The run times you will get for your queries will vary based on what portion of the data is cached. If you want to make credible performance comparisons in terms of run times, make sure that the caching environment in both cases reflect what you would have in your production environment. That is, if you expect most pages to reside in cache in your production environment (warm cache), run each query twice, and measure the run time of the second run. If you expect most pages not to reside in cache (cold cache), in your tests clear the cache before you run each query.

Before you proceed, drop the index that you just created:

```
DROP INDEX dbo.Orders.idx_nc_od_i_oid_cid_eid_sid;
```

Unordered Nonclustered Index Scan + Lookups

The next step in our index optimization scale is to create a smaller nonclustered index that would not cover the query and that contains the filtered column (*orderid*), but not as the first key column:

```
CREATE NONCLUSTERED INDEX idx_nc_od_i_oid
   ON dbo.Orders(orderdate)
   INCLUDE(orderid);
```

You would get an unordered nonclustered index scan + lookups as shown in Figure 3-49:

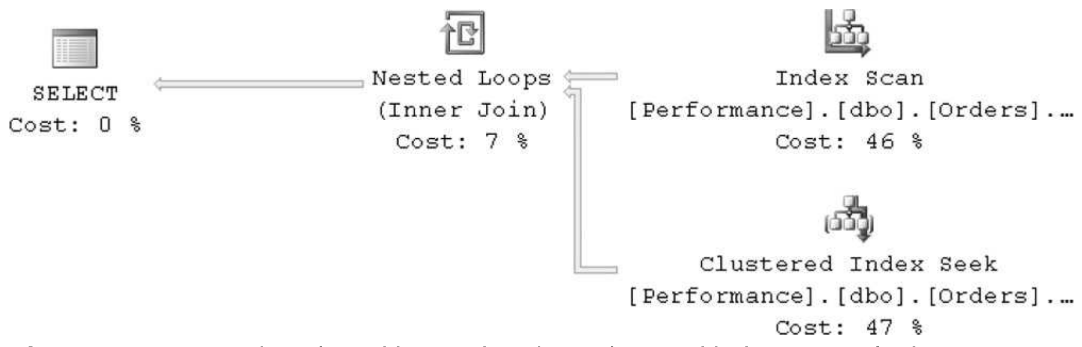


Figure 3-49: Execution plan with unordered nonclustered index scan + lookups

Note that the efficiency of this plan compared to the previous one will depend on the selectivity of the query. As the selectivity of the query grows larger, the more substantial the cost is of the lookups here. In our case, the query is fairly selective, so this plan is more efficient than the previous two; however, with low selectivity, this plan will be less efficient than the previous two.

Here are the performance measures that I got for this query:

- Logical reads: 5923
- CPU time: 100 ms
- Elapsed time: 379 ms
- Estimated subtree cost: 7.02136

Note that even though the number of logical reads and the query cost seem higher than in the previous plan, you can see that the run times are lower. Remember that the lookup operations here traverse the clustered index, and the nonleaf levels of the clustered index are most likely to reside in cache.

Before you continue, drop the new index:

```
DROP INDEX dbo.Orders.idx_nc_od_i_oid;
```

Nonclustered Index Seek + Ordered Partial Scan + Lookups

You can get the next level of optimization in the scale by creating a nonclustered noncovering index on *orderid:*

```
CREATE UNIQUE NONCLUSTERED INDEX idx_unc_oid
  ON dbo.Orders(orderid);
```

This index would yield a nonclustered index seek + ordered partial scan + lookups as shown in Figure 3-50.

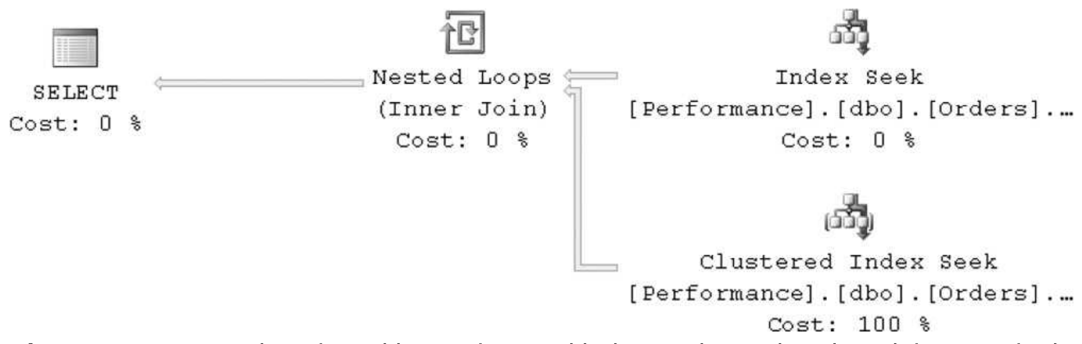


Figure 3-50: Execution plan with nonclustered index seek + ordered partial scan + lookups

Instead of performing the full index scan as the previous plan did, this plan performs a seek to the first key in the sought range, followed by an ordered partial scan of only the relevant range. Still, you get as many lookups as previously, which in our case amounts to a big chunk of the query cost. As the range grows larger, the contribution of the lookups to the query's cost becomes more substantial, and the costs of these two plans would become closer and closer.

Here are the performance measures for this query:

- Logical reads: 3077
- CPU time: 0 ms
- Elapsed time: 53 ms
- Estimated subtree cost: 3.22852

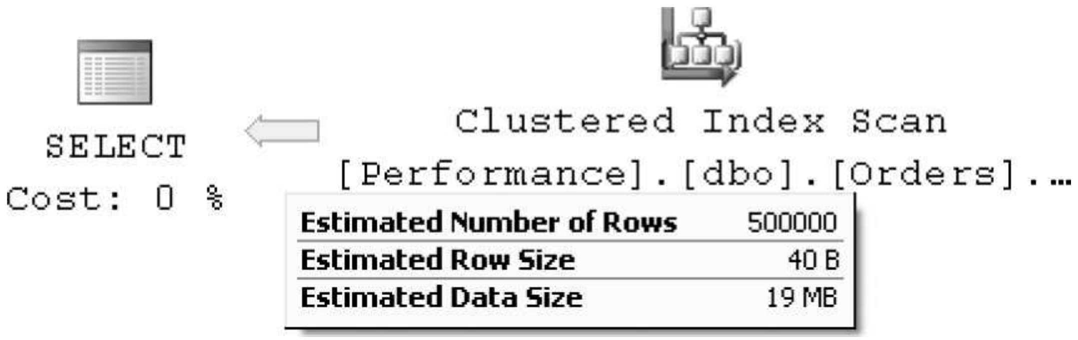
Determining the Selectivity Point

Allow me to digress a bit to expand on a subject I started discussing earlier—plans that are dependent on the selectivity of the query. The efficiency of the last plan is dependent on selectivity because you get one whole lookup per sought row. At some selectivity point, the optimizer would realize that a table scan is more efficient than using this plan. You might find it surprising, but that selectivity point is a pretty small percentage. Even if you have no clue about how to calculate this point, you can practice a trial-and-error approach, where you apply a binary algorithm, shifting the selectivity point to the left or right based on the plan that you get. Remember that *high selectivity* means a low percentage of rows, so going to the left of a selectivity point (lowering the percentage) would mean getting higher selectivity. You can invoke a range query, where you start with 50 percent selectivity by invoking the following query:

```
SELECT orderid, custid, empid, shipperid, orderdate
```

```
FROM dbo.Orders
WHERE orderid >= 500001;
```

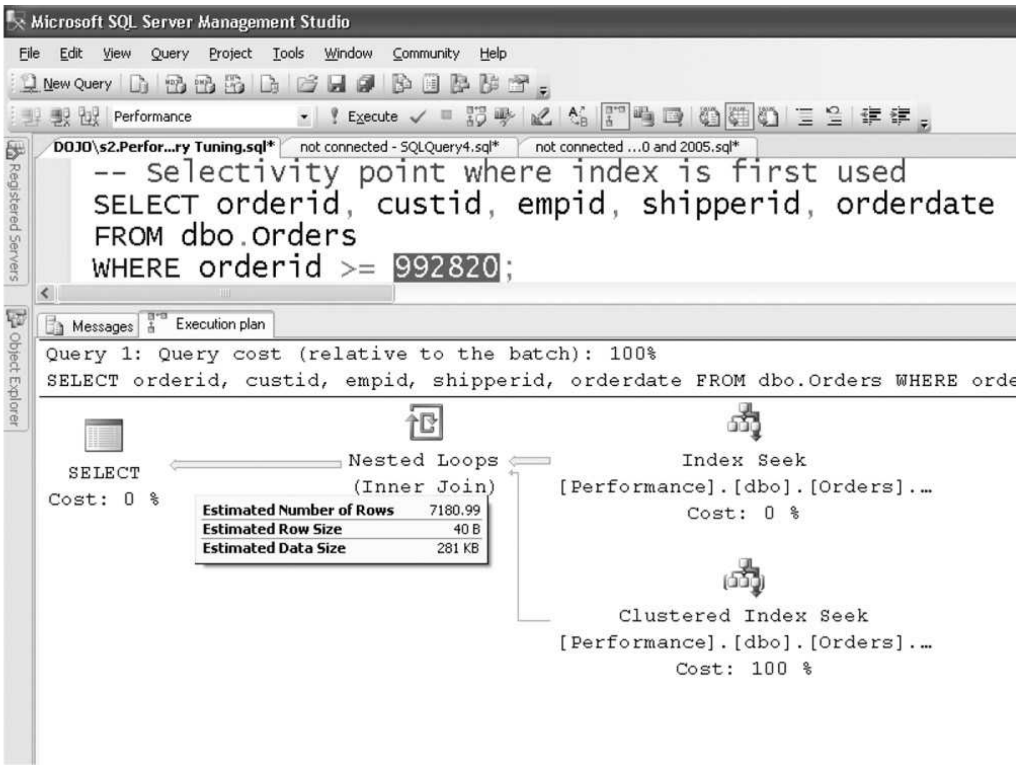
Examine the estimated (no need for actual here) execution plan, and determine whether to proceed in the next step to the left or to the right of this point, based on whether you got a table scan (clustered index scan) or an index seek. With the median key, you get the plan shown in Figure 3-51, showing a table scan:



Estimated Number of Rows	500000
Estimated Row Size	40 B
Estimated Data Size	19 MB

Figure 3-51: Estimated plan showing a table scan

This tells you that 50 percent is not selective enough to justify using the nonclustered index. So you go to the right, to the middle point between 50 percent and a 100 percent. Following this logic, you would end up using the following keys: 750001, 875001, 937501, 968751, 984376, 992189, 996095. The last key would yield a plan where the nonclustered index is used. So now you go to the left, to the point between the keys 992189 and 996095, which is 994142. You will find that the nonclustered index is still used, so you keep on going left, to the point between the keys 992189 and 994142. You continue this process, going left or right according to your findings, until you reach the first selectivity point where the nonclustered index is used. You will find that this point is the key 992820, producing the plan shown in Figure 3-52:


Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Community Help
New Query
Performance
Execute
DOJO\s2.Perfor...ry Tuning.sql*
not connected - SQLQuery4.sql*
not connected ...0 and 2005.sql*
Registered Servers
Object Explorer
Messages
Execution plan

```
-- Selectivity point where index is first used
SELECT orderid, custid, empid, shipperid, orderdate
FROM dbo.Orders
WHERE orderid >= 992820;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT orderid, custid, empid, shipperid, orderdate FROM dbo.Orders WHERE orde
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Index Seek
[Performance].[dbo].[Orders].…
Cost: 0 %
Clustered Index Seek
[Performance].[dbo].[Orders].…
Cost: 100 %

Estimated Number of Rows	7180.99
Estimated Row Size	40 B
Estimated Data Size	281 KB

Figure 3-52: Estimated plan showing the index is used

You can now calculate the selectivity, which is the number of qualifying rows (7181) divided by the number of rows in the table (1,000,000), which amounts to 0.7181 percent.

In our query pattern's case, with this selectivity or higher (lower percentage), the optimizer will use the nonclustered index, while with a lower selectivity, it will opt for a table scan. As you can see, in our query pattern's case, the selectivity point is even lower than one percent. Some database professionals might find this number surprisingly small, but if you make performance estimations like the ones we did earlier, you will find it reasonable. Don't forget that page reads is not the only factor that you should take into consideration. You should also consider the access pattern (random/sequential) and other factors as well. Remember that random I/O is much more expensive than sequential I/O. Lookups use random I/O, while a table scan uses sequential I/O.

Before you proceed, drop the index used in the previous step:

```
DROP INDEX dbo.Orders.idx_unc_oid;
```

Clustered Index Seek + Ordered Partial Scan

You can get the next level of optimization by creating a clustered index on the *orderid* column. Because there's already a clustered index on the Orders table, drop it first and then create the desired one:

```
DROP INDEX dbo.Orders.idx_cl_od;
CREATE UNIQUE CLUSTERED INDEX idx_cl_oid ON dbo.Orders(orderid);
```

You will get a trivial plan that uses a seek to the first key matching the filter, followed by an ordered partial scan of the sought range, as shown in Figure 3-53.

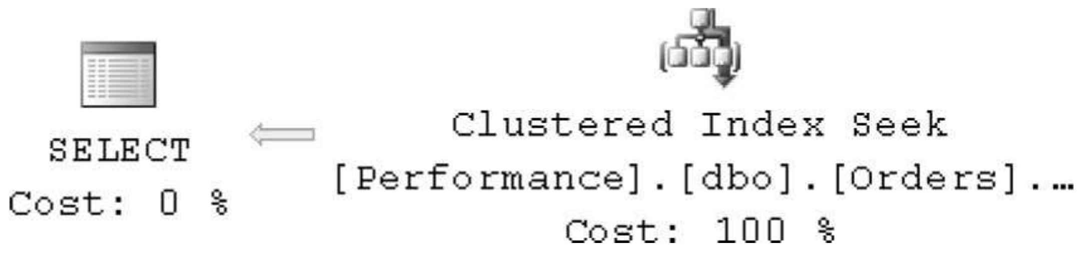


Figure 3-53: Execution plan with clustered index seek + ordered partial scan

The main benefit of this plan is that there are no lookups involved. As the selectivity of the query goes lower, this plan becomes more and more efficient compared to a plan that does apply lookups. The I/O cost involved with this plan is the cost of the seek (3 in our case), plus the number of pages that hold the data rows in the filtered range (26 in our case). The main cost of such a plan is typically, for the most part, the cost of the ordered partial scan, unless the range is really tiny (for example, a point query). Remember that the performance of an ordered index scan will, to a great extent, depend on the fragmentation level of the index. Here are the performance measures that I got for this query:

- Logical reads: 29
- CPU time: 0 ms
- Elapsed time: 87 ms
- Estimated subtree cost: 0.022160

Before proceeding to the next step, restore the original clustered index:

```
DROP INDEX dbo.Orders.idx_cl_oid;
CREATE UNIQUE CLUSTERED INDEX idx_cl_od ON dbo.Orders(orderid);
```

Covering Nonclustered Index Seek + Ordered Partial Scan

The optimal level in our scale is a nonclustered covering index defined with the *orderid* column as the key and all the other columns as included non-key columns:

```
CREATE UNIQUE NONCLUSTERED INDEX idx_unc_oid_i_od_cid_eid_sid
  ON dbo.Orders(orderid)
  INCLUDE(orderdate, custid, empid, shipperid);
```

The plan's logic is similar to the previous one, only here, the ordered partial scan ends up reading fewer pages. That, of course, is because more rows fit in a leaf page of this index than data rows do in a clustered index page. You get the plan shown in Figure 3-54.

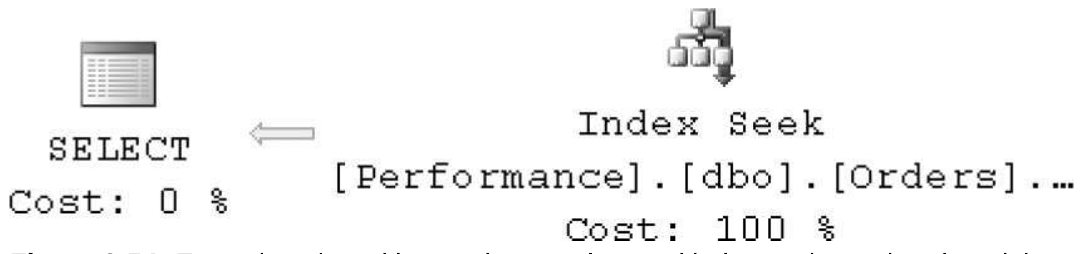


Figure 3-54: Execution plan with covering nonclustered index seek + ordered partial scan

And here are the performance measures I got for this query:

- Logical reads: 9
- CPU time: 0 ms
- Elapsed time: 47 ms
- Estimated subtree cost: 0.008086

Again, this is a trivial plan. And also here, the performance of the ordered partial scan will vary depending on the fragmentation level of the index. As you can see, the cost of the query dropped from 19.621100 in the lowest level in the scale to 0.008086, and the elapsed time from more than 16 seconds to 47 milliseconds. Such a drop in run time is common when tuning indexes in an environment with poor index design.

When done, drop the last index you created:

```
DROP INDEX dbo.Orders.idx_unc_oid_i_od_cid_eid_sid;
```

Index Optimization Scale Summary and Analysis

Remember that the efficiency of several plans in our index optimization scale was based on the selectivity of the query. If the selectivity of a query you're tuning varies significantly between invocations of the query, make sure that in your tuning process you take this into account. For example, you can prepare tables and graphs with the performance measurements vs. selectivity, and analyze such data before you make your index design choices. For example, Table 3-16 shows a summary of logical reads vs. selectivity of the different levels in the scale for the sample query pattern under discussion against the sample Orders table.

Table 3-16: Logical Reads vs. Selectivity for Each Access Method

Access Method	1	1,000	10,000	100,000	200,000	500,000	1,000,000	rows
	0.0001%	0.1%	1%	10%	20%	50%	100%	selectivity
Table Scan/Unordered Clustered Index Scan	25,080	25,080	25,080	25,080	25,080	25,080	25,080	
Unordered Covering Nonclustered Index Scan	5,095	5,095	5,095	5,095	5,095	5,095	5,095	
Unordered Covering Nonclustered Index Scan + Lookups	2,855	5,923	33,486	309,111	615,361	1,534,111	3,065,442	
Nonclustered Index Seek + Ordered Partial Scan + Lookups	6	3,078	30,667	306,547	613,082	1,532,685	3,073,741	
Clustered Index Seek + Ordered Partial Scan	4	29	249	2,447	4,890	12,219	24,433	
Covering Nonclustered Index Seek + Ordered Partial Scan	4	9	54	512	1,021	2,546	5,087	

Note To apply a certain execution plan in a case where the optimizer would normally opt for another plan that is more efficient, I had to use a table hint to force using the relevant index.

Of course, logical reads shouldn't be the only indication you rely on. Remember that different I/O patterns have different performance, and that physical reads are much more expensive than logical reads. But when you see a significant difference in logical reads between two options, it is usually a good indication of which option is faster. Figure 3-55 has a graphical depiction of the information from Table 3-16.

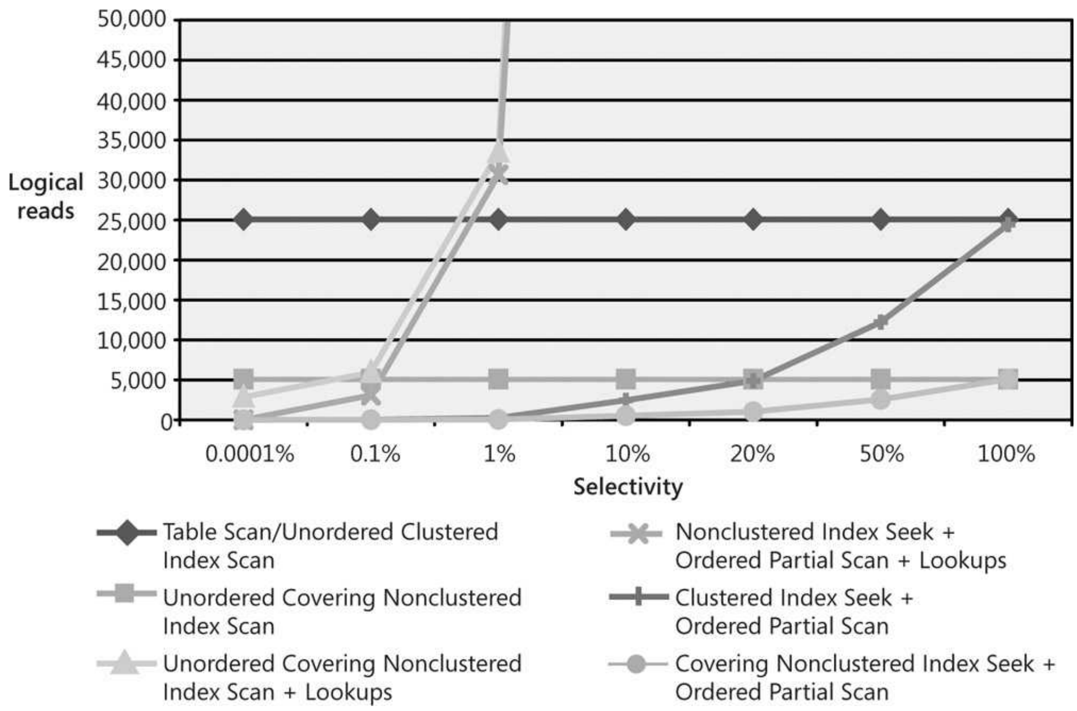


Figure 3-55: Graph of logical reads vs. selectivity

You can observe many interesting things when analyzing the graph. For example, you can clearly see which plans are based on selectivity and which aren't. And also, you can see the selectivity point at which one plan becomes better than another.

Similarly, Table 3-17 shows summary performance statistics of the query cost vs. selectivity.

Table 3-17: Estimated Subtree Costs vs. Selectivity for Each Access Method

Access Method	1	1,000	10,000	100,000	200,000	500,000	1,000,000	rows
	0.0001%	0.1%	1%	10%	20%	50%	100%	selectivity
Table Scan/Unordered Clustered Index Scan	19.621100	19.621100	19.621100	19.621100	19.621100	19.621100	19.621100	
Unordered Covering Nonclustered Index Scan	4.863280	4.863280	4.863280	4.863280	4.863280	4.863280	4.863280	
Unordered Covering Nonclustered Index Scan + Lookups	3.690270	7.021360	30.665400	96.474000	113.966000	163.240000	244.092000	
Nonclustered Index Seek + Ordered Partial Scan + Lookups	0.006570	3.228520	21.921600	100.881000	127.376000	204.329000	335.109000	
Clustered Index Seek + Ordered Partial Scan	0.022160	0.022160	0.022160	0.022160	0.022160	0.022160	19.169900	
Covering Nonclustered Index Seek + Ordered Partial Scan	0.008086	0.008086	0.008086	0.008086	0.008086	0.008086	4.862540	

Figure 3-56 shows a graph based on the data in Table 3-17.

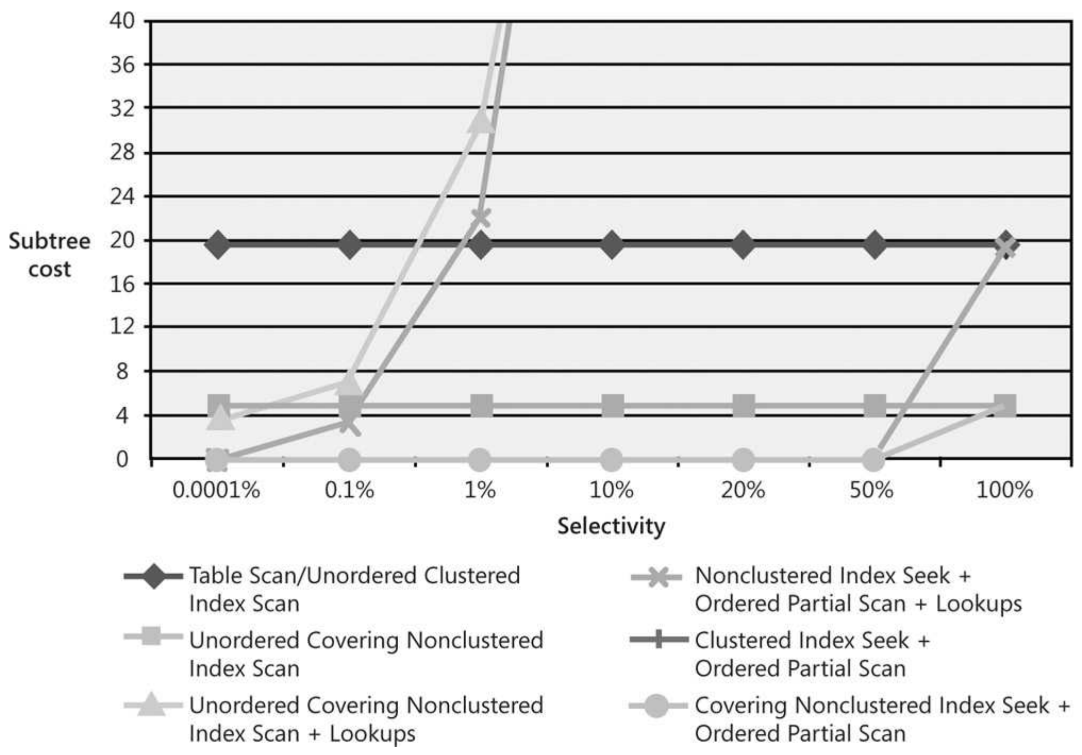


Figure 3-56: Graph of subtree cost vs. selectivity

You can observe a striking resemblance between the two graphs; but when you think about it, it makes sense because most of the cost involved with our query pattern is because of I/O.

Naturally, in plans where a more substantial portion of the cost is related to CPU, you will get different results.

Of course, you would also want to generate similar statistics and graphs for the actual run times of the queries in your benchmarks. At the end of the day, run time is what the user cares about.

I also find it valuable to visualize performance information in another graphical way as shown in Figure 3-57.

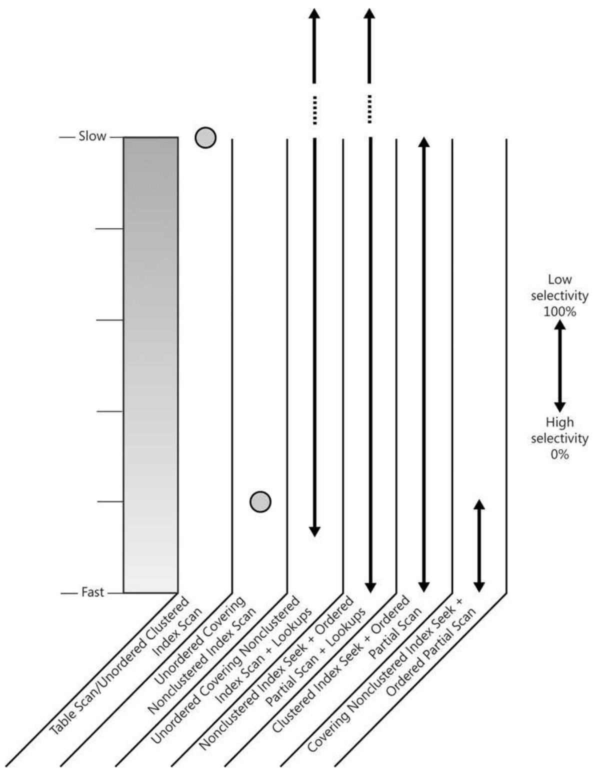

Slow
Fast
Low selectivity 100%
High selectivity 0%
Table Scan/Unordered Clustered Index Scan
Unordered Covering Nonclustered Index Scan
Unordered Covering Nonclustered Index Scan + Lookups
Nonclustered Index Seek + Ordered Partial Scan + Lookups
Clustered Index Seek + Ordered Partial Scan
Covering Nonclustered Index Seek + Ordered Partial Scan

Figure 3-57: Index optimization scale

You might find it easier with this illustration to identify plans that are based on selectivity vs. plans that aren't (represented as a dot), and also to make comparisons between the performance of the different levels of optimization in the scale.

Note For simplicity's sake, all statistics and graphs shown in this section were collected against the Performance database I used in this chapter, where the level of fragmentation of indexes was minimal. When you conduct benchmarks and performance tests, make sure you introduce the appropriate levels of fragmentation in the indexes in your test system so that they reflect the fragmentation levels of the indexes in your production system adequately. The performance of ordered scans might vary significantly based on the level of fragmentation of your indexes. Similarly, you also need to examine the average page densities in your production system, and introduce similar page densities in the test system.

Besides having the ability to design good indexes, it is also important to be able to identify which indexes are used more heavily and which are rarely or never used. You don't want to keep indexes that are rarely used, as they do have negative performance effects on modifications. In SQL Server 2000, the only available techniques to determine index usage were very awkward, to say the least. For example, you could take a script that contains a representative sampling of the queries that run in the system and produce textual SHOWPLAN information for the queries in the script. You could write an application that parses the output, extracts the names of the indexes that are used, and calculates index usage statistics. Obviously, it's a problematic technique that also requires a fair amount of effort.

Another option you could use in SQL Server 2000 is to run the Index Tuning Wizard (ITW) based on a sample workload. Among the reports that ITW generates is a report that shows you usage statistics of the existing indexes for the input queries.

In SQL Server 2005, you have much more powerful and convenient tools to work with. You get a DMF called *dm_db_index_operational_stats* and a DMV called *dm_db_index_usage_stats.* The *dm_db_index_operational_stats* DMF gives you low-level I/O, locking, latching, and access method activity information. You provide the function a database ID, object ID, index ID (or 0 for a heap), and partition ID. You can also request information about multiple entities by specifying a NULL in the relevant argument. For example, to get information about all objects, indexes, and partitions in the Performance database, you would invoke the function as follows:

```
SELECT *
FROM sys.dm_db_index_operational_stats(
  DB_ID('Performance'), null, null, null);
```

The *dm_db_index_usage_stats* DMV gives you usage counts of the different index operations:

```
SELECT *
FROM sys.dm_db_index_usage_stats;
```

These dynamic management objects make the analysis of index usage much simpler and more accurate than before.

Fragmentation

I referred to index fragmentation on multiple occasions in this chapter. When I mentioned fragmentation, I referred to a type known as *logical scan fragmentation* or *average fragmentation in percent* or *external fragmentation.* As I mentioned earlier, this type reflects the percentage of out-of-order pages in the index, in terms of their physical order vs. their logical order in the linked list. Remember that this fragmentation can have a substantial impact on ordered scan operations in indexes. It has no effect on operations that do not rely on the index's linked list— for example, seek operations, lookups, unordered scans, and so on. You want to minimize the fragmentation level of indexes for queries with a substantial portion of their cost involved with ordered scans. You do so by rebuilding or reorganizing indexes.

Another type of fragmentation that you typically care about is what I referred to as average page density. Some database professionals refer to this type of fragmentation as *internal fragmentation*, but to avoid confusion I consciously didn't use this term earlier. While logical scan fragmentation is never a good thing, average scan fragmentation has two facets. A low percentage (low level of page population) has a negative impact on queries that read data, as they end up reading more pages than they could potentially if the pages were better populated. The positive impact of having some free space in index pages is that insertions of rows to such pages would not cause page splits—which are very expensive. As you can guess, free space in index pages is bad in systems that involve mostly reads (for example, data warehouses) and good for systems that involve many inserts (for example, OLTP systems). You might even want to introduce some free space in index pages by specifying a fillfactor value when you rebuild your indexes.

To determine whether you need to rebuild or reorganize your indexes, you need information about both types of fragmentation. In SQL Server 2005, you can get this information by querying the DMF *dm_db_index_physical_stats.* For example, the following query will return fragmentation information about the indexes in the Performance database:

```
SELECT *
FROM sys.dm_db_index_physical_stats(
  DB_ID('Performance'), NULL, NULL, NULL, NULL);
```

The fragmentation types I mentioned will show up in the attributes *avg_fragmentation_in_ percent* and *avg_page_space_used_in_percent*, and as you can see, the attribute names are self explanatory. In SQL Server 2000, you use the DBCC SHOWCONTIG command to get similar information:

```
DBCC SHOWCONTIG WITH ALL_INDEXES, TABLE RESULTS, NO_INFOMSGS;
```

The two attributes of interest to our discussion are *LogicalFragmentation* and *AveragePageDensity.*

As I mentioned earlier, to treat both types of fragmentation you need to rebuild or reorganize the index. Rebuilding an index has the optimal defragmentation effect. The operation makes its best attempt to rebuild the index in the same physical order on disk as in the linked list and to make the pages as contiguous as possible. Also, remember that you can specify a fillfactor to introduce some free space in the index leaf pages.

In SQL Server 2000, index rebuilds were offline operations. Rebuilding a clustered index acquired an exclusive lock for the whole duration of the operation, meaning that process can neither read nor write to the table. Rebuilding a nonclustered index acquired a shared lock, meaning that writes are blocked against the table, and obviously, the index can not be used during the operation. SQL Server 2005 introduces *online index operations* that allow you to create, rebuild, and drop indexes online. In addition, these operations allow users to interact with the data while the operation is in progress. Online index operations use the new rowversioning technology introduced in SQL Server 2005. When an index is rebuilt online, SQL Server actually maintains two indexes behind the scenes, and when the operation is done, the new one overrides the old one.

As an example, the following code rebuilds the *idx_cl_od* index on the Orders table online:

```
ALTER INDEX idx_cl_od ON dbo.Orders REBUILD WITH (ONLINE = ON);
```

Note that online index operations need sufficient space in the database and overall are slower than offline operations. If you can spare a maintenance window for the activity to work offline, you better do so. Even when you do perform the operations online, they will have a performance impact on the system while they are running, so it's best to run them during off-peak hours.

Instead of rebuilding an index, you can also reorganize it. Reorganizing an index involves a bubble sort algorithm to sort the index pages physically on disk according to their order in the index's linked list. The operation does not attempt to make the pages more contiguous (reduce gaps). As you can guess, the defragmentation level that you get from this operation is not as optimal as fully rebuilding an index. Also, overall, this operation performs more logging than an index rebuild and, therefore, is typically slower.

So why use this type of defragmentation? First, in SQL Server 2000 it was the only online defragmentation utility. The operation grabs short-term locks on a pair of pages at a time to determine whether they are in the correct order, and if they are not, it swaps them. Second, an index rebuild must run as a single transaction, and if it's aborted while it's in process, the whole activity is rolled back. This is unlike an index reorganize operation, which can be interrupted as it operates on a pair of pages at a time. When you later run the reorganize activity again, it will pick up where it left off earlier.

Here's how you would reorganize the *idx_cl_od* index in SQL Server 2005:

```
ALTER INDEX idx_cl_od ON dbo.Orders REORGANIZE;
```

In SQL Server 2000, you use the DBCC INDEX DEFRAG command for the same purpose.

Partitioning

SQL Server 2005 introduces native partitioning of tables and indexes. Partitioning your objects means that they are internally split into multiple physical units that together make the object (table or index). Partitioning is virtually unavoidable in medium to large environments. By partitioning your objects, you improve the manageability and maintainability of your system and you improve the performance of activities such as purging historic data, data loads, and others as well. Partitioning in SQL Server is native—that is, you have built-in tools to partition the tables and indexes, while logically, to the applications and users they appear as whole units. In SQL Server 2000, partitioning was achieved by manually creating multiple tables and a view on top that unifies the pieces. SQL Server 2005 native partitioning has many advantages over partitioned views in SQL Server 2000, including improved execution plans, a substantially relaxed set of requirements, and more.

More Info Partitioning is outside the scope of this book, but I covered it in detail in a series of articles in *SQL Server Magazine.* You can find these articles at http://www.sqlmag.com. (Look up instant document IDs 45153, 45533, 45877, and 46207.)

Preparing Sample Data

When conducting performance tests, it is vital that the sample data you use be well prepared so that it reflects the production system as closely as possible, especially with respect to the factors you are trying to tune. Typically, it's not realistic to just copy all the data from the production tables, at least not with the big ones. However, you should make your best effort to have an adequate representation, which will reflect similar data distribution, density of keys, cardinality, and so on. Also, you want your queries against the test system to have similar selectivity to the queries against the production system. Performance tests can be skewed when the sample data does not adequately represent the production data.

In this section, I'll provide an example of skewed performance testing results resulting from inadequate sample data. I'll also discuss the new TABLE SAMPLE option.

Data Preparation

When I prepared the sample data for this chapter's demonstrations, I didn't need to reflect a specific production system, so preparing sample data was fairly simple. I mainly needed it for the "Tuning Methodology" and "Index Tuning" sections. I could express most of my points through simple random distribution of the different attributes that were relevant to our discussions. But our main data table—Orders—does not accurately reflect an average production Orders table. For example, I produced a fairly even distribution of values in the different attributes, while typically in production systems, different attributes have different types of distribution (some uniform, some standard). Some customers place many orders, and others place few. Also, some customers are more active in some periods of time and less active in others. Depending on your tuning needs, you might or might not need to reflect such things in your sample data, but you definitely need to consider them and decide whether they do matter.

When you need large tables with sample data, the easiest thing to do is to generate some small table and duplicate its content (save the key columns) many times. This can be fine if, for example, you want to test the performance of a user-defined function invoked against every row or a cursor manipulation iterating through many rows. But such sample data in some cases can yield completely different performance than what you would get with sample data that more adequately reflects your production data. To demonstrate this, I'll walk you through an example that I cover in much more depth in *Inside T-SQL Programming.* I often give this exercise in class and ask students to prepare a large amount of sample data without giving any hints.

The exercise has to do with a table called Sessions, which you create and populate by running the code in Listing 3-6.

Listing 3-6: Creating and populating the Sessions table

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

CREATE TABLE dbo.Sessions
(
  keycol      INT          NOT NULL IDENTITY,
  app         VARCHAR(10)  NOT NULL,
  usr         VARCHAR(10)  NOT NULL,
  host        VARCHAR(10)  NOT NULL,
  starttime   DATETIME     NOT NULL,
  endtime     DATETIME     NOT NULL,
  CONSTRAINT  PK_Sessions  PRIMARY KEY(keycol),
  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', '2003021209:30');
INSERT INTO       dbo.Sessions
  VALUES('app1', 'user4', 'host2', '20030212  09:15', '2003021210:30');
INSERT INTO       dbo.Sessions
  VALUES('app1', 'user5', 'host3', '20030212  09:15', '2003021209:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user6', 'host3', '20030212  10:30', '2003021214:30');
INSERT INTO dbo.Sessions
  VALUES('app1', 'user7', 'host4', '20030212  10:45', '20030212 11:30');
INSERT INTOdbo.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_nc_app_st_et ON dbo.Sessions(app, starttime, endtime);
```

The Sessions table contains information about user sessions against different applications. The request is to calculate the maximum number of concurrent sessions per application—that is, the maximum number of sessions that were active at any point in time against each application.

The following query produces the requested information, as shown in Table 3-18:

Table 3-18: Max Concurrent Sessions per Application

app	mx
app1	4
app2	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;
```

The derived table S1 contains the distinct application name (*app*) and session start time (*starttime as ts*) pairs. For each row of S1, a subquery counts the number of sessions that were active for the application *S1.app* at time *S1.ts.* The outer query then groups the data by *app* and returns the maximum count for each group. SQL Server's optimizer generates the execution plan shown in Figure 3-58 for this query.

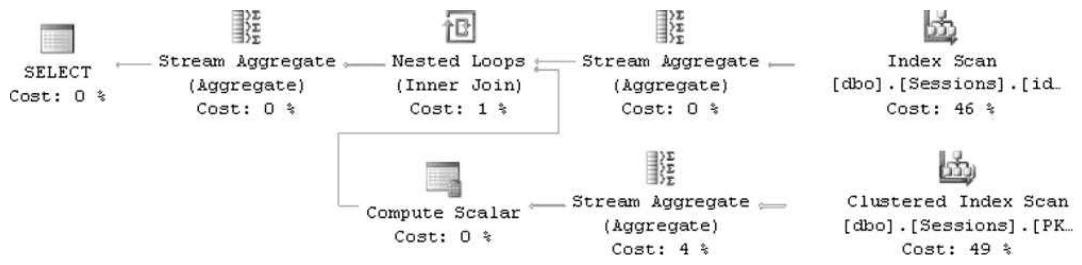


Figure 3-58: Execution plan for query against the Sessions table

In the script in Listing 3-6, I created the covering index *idx_nc_app_st_et* on (*app, starttime, endtime*), which is the optimal index for this query. In the plan, this index is scanned in order (Index Scan operator) and distinct rows are isolated (Stream Aggregate operator). As rows are streamed out from the Stream Aggregate operator, a Nested Loops operator invokes a series of activities to calculate the count of active sessions for each row. Because the Sessions table is so tiny (only one page of data), the optimizer simply decides to scan the whole table (unordered clustered index scan) to calculate each count. With a larger data set, instead of scanning the table, the plan would perform a seek and ordered partial scan of the covering index to obtain each count. Finally, another Stream Aggregate operator groups the data by *app* to calculate the maximum count for each group.

Now that you're familiar with the problem, suppose you were asked to prepare sample data with 1,000,000 rows in the source table (call it BigSessions) such that it would represent a realistic environment. Ideally, you should be thinking about the number of customers, the number of different order dates, and so on. However, people often take the most obvious approach, which is to duplicate the data from the small source table many times; in our case, such an approach would drastically skew the performance compared to a more realistic representation of production environments.

Now run the code in Listing 3-7 to generate the BigSessions table by duplicating the data from the Sessions table many times. You will get 1,000,000 rows in the BigSessions table.

Listing 3-7: Populate sessions with inadequate sample data

```
SET NOCOUNT ON;
USE Performance;
GO
IF OBJECT_ID('dbo.BigSessions')IS NOT NULL
```

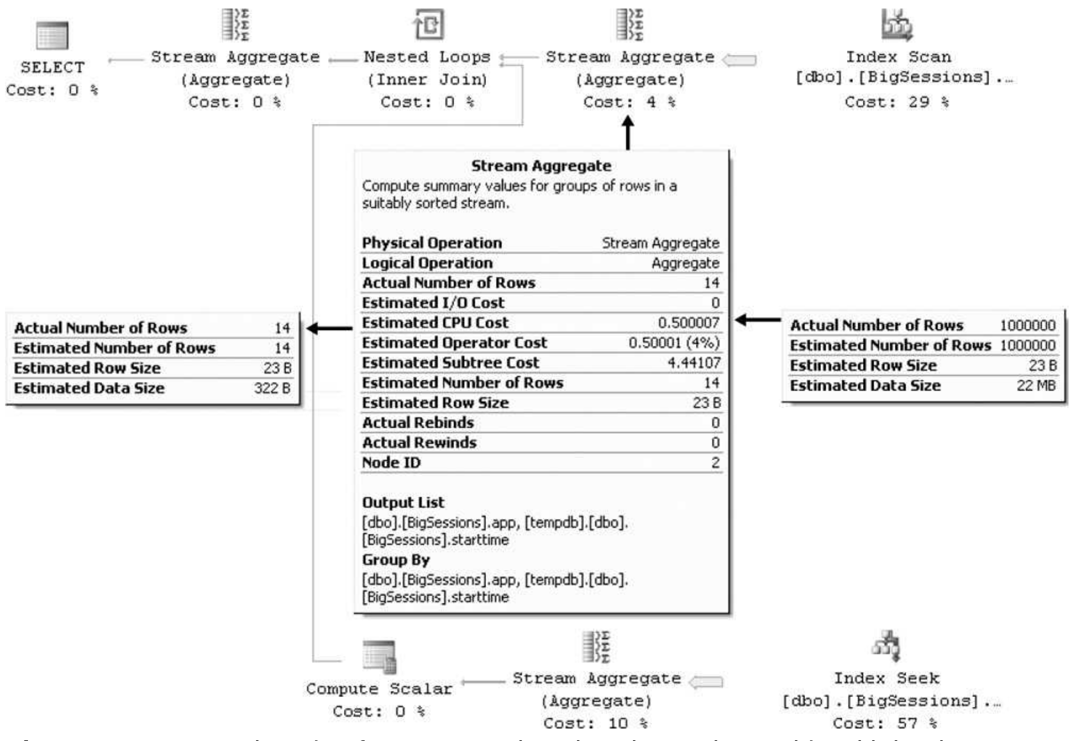
```
  DROP TABLE dbo.BigSessions;
GO
SELECT IDENTITY(int, 1, 1) AS keycol,
  app, usr, host, starttime, endtime
INTO dbo.BigSessions
FROM dbo.Sessions AS S, Nums
WHERE n <= 62500;

ALTER TABLE dbo.BigSessions
  ADD CONSTRAINT PK_BigSessions PRIMARY KEY(keycol);
CREATE INDEXidx_nc_app_st_et
  ON dbo.BigSessions(app, starttime, endtime);
```

Run the following query against BigSessions:

```
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app,
        (SELECT COUNT(*)
         FROM dbo.BigSessions 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.BigSessions) AS S1) AS C
GROUP BY app;
```

Note that this is the same query as before (but against a different table). The query will finish in a few seconds, and you will get the execution plan shown in Figure 3-59.



Stream Aggregate
Compute summary values for groups of rows in a suitably sorted stream.

Physical Operation	Stream Aggregate
Logical Operation	Aggregate
Actual Number of Rows	14
Estimated I/O Cost	0
Estimated CPU Cost	0.500007
Estimated Operator Cost	0.50001 (4%)
Estimated Subtree Cost	4.44107
Estimated Number of Rows	14
Estimated Row Size	23 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	2

Output List
[dbo].[BigSessions].app, [tempdb].[dbo].[BigSessions].starttime
Group By
[dbo].[BigSessions].app, [tempdb].[dbo].[BigSessions].starttime

Actual Number of Rows	14
Estimated Number of Rows	14
Estimated Row Size	23 B
Estimated Data Size	322 B

Actual Number of Rows	1000000
Estimated Number of Rows	1000000
Estimated Row Size	23 B
Estimated Data Size	22 MB

Figure 3-59: Execution plan for query against the BigSessions table with inadequate sample data

Here are the performance measures I got for this query:

- Logical reads: 20024
- CPU time: 1452 ms
- Elapsed time: 1531 ms
- Estimated subtree cost: 13.6987

Of course, after the first Stream Aggregate operator eliminated duplicate rows, it yielded only 14 distinct rows; so the Nested Loops operator that follows invoked only 14 iterations of the activity that calculates the count of active sessions. Therefore, the query finished in only a few seconds. In a production environment, there might be only a few applications, but so few distinct start times would be unlikely.

You might think it's obvious that the sample data is inadequate because the first step in the query constructing the derived table S1 isolates distinct *app* and *starttime* values—meaning that the COUNT aggregate in the subquery will be invoked only a small number of times. But in many cases, the fact that there's a performance skew due to bad sample data is much more elusive. For example, many programmers would come up with the following query solution to our problem, where they won't isolate distinct *app* and *starttime* values first:

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

Examine the execution plan generated for this query, as shown in Figure 3-60.

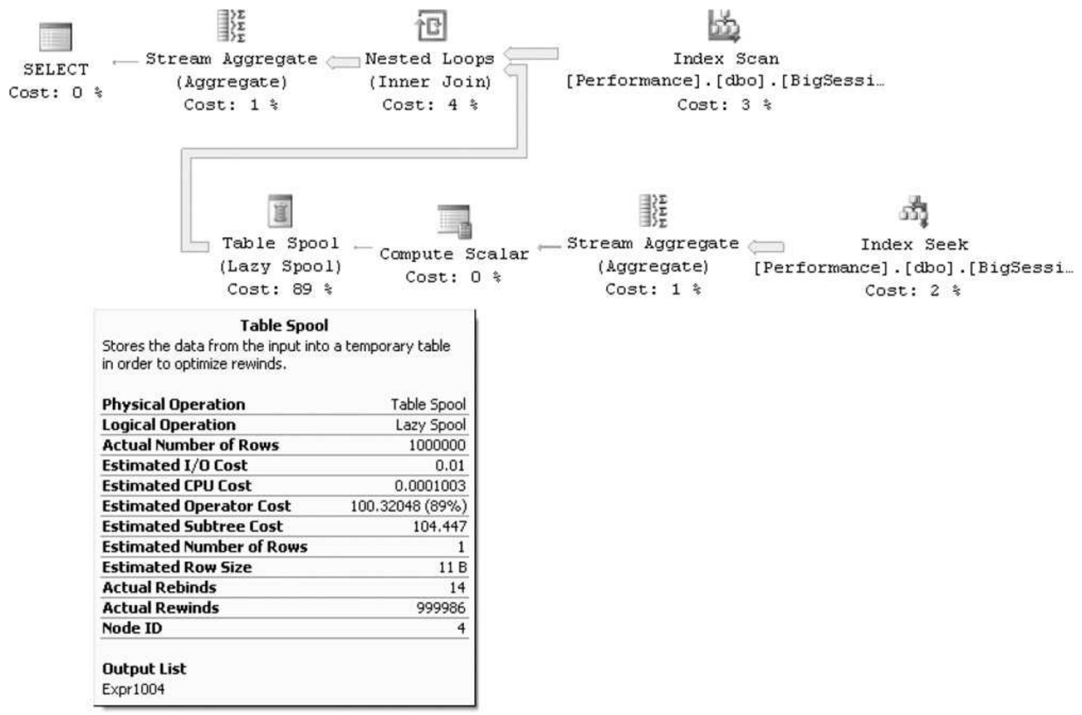


Table Spool

Stores the data from the input into a temporary table in order to optimize rewinds.

Physical Operation	Table Spool
Logical Operation	Lazy Spool
Actual Number of Rows	1000000
Estimated I/O Cost	0.01
Estimated CPU Cost	0.0001003
Estimated Operator Cost	100.32048 (89%)
Estimated Subtree Cost	104.447
Estimated Number of Rows	1
Estimated Row Size	11 B
Actual Rebinds	14
Actual Rewinds	999986
Node ID	4

Output List

Expr1004

Figure 3-60: Execution plan for query against the BigSessions table with inadequate sample data

Focus on the Table Spool operator, which represents a temporary table holding the session count for each distinct combination of *app* and *starttime* values. Because you didn't isolate distinct *app* and *starttime* values, the optimizer elegantly does the job for you. Notice the number of rebinds (14) and the number of rewinds (999,986). Remember that a rebind means that one or more correlated parameters of the join operator changed and the inner side must be reevaluated. That happens 14 times, once for each distinct pair of *app* and *starttime*—meaning that the actual count activity preceding the operator took place only 14 times. A rewind means that none of the correlated parameters changed and the prior inner result set can be reused; this happened 999,986 times (1,000,000–14 = 999,986). Naturally, with more realistic data distribution for our scenario, the count activity will take place many more times than 14, and you will get a much slower query. It was a mistake to prepare the sample data by simply copying the rows from the small Sessions table many times. The distribution of values in the different columns should represent production environments more realistically.

Run the code in Listing 3-8 to populate BigSessions with more adequate sample data.

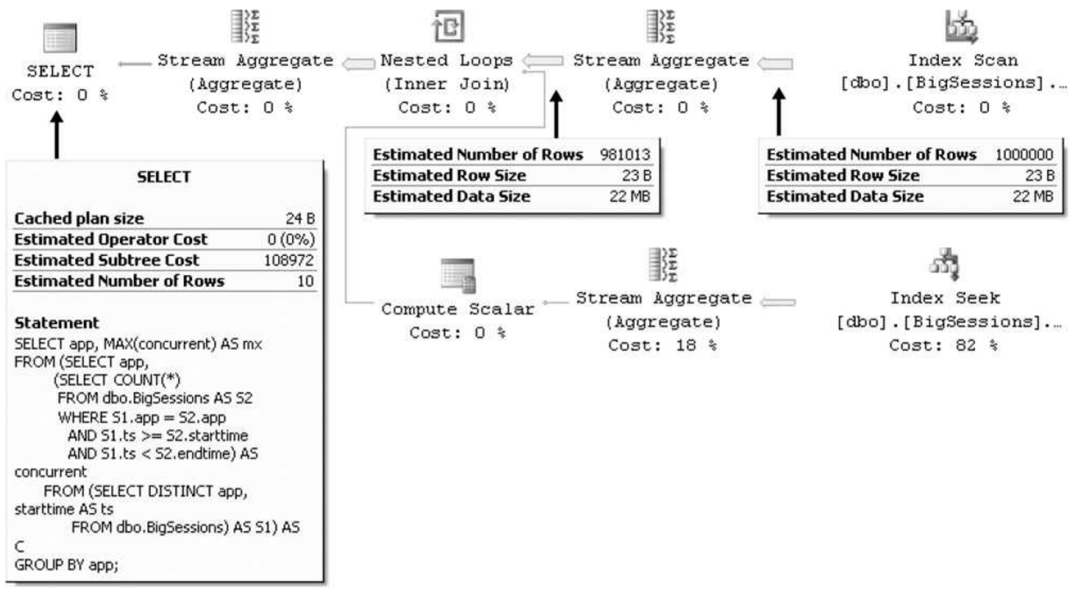
Listing 3-8: Populate the BigSessions table with adequate sample data

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

SELECT
  IDENTITY(int, 1, 1) AS keycol,
  D.*,
  DATEADD(
    second,
    1 + ABS(CHECKSUM(NEWID())) % (20*60),
    starttime) AS endtime
INTO dbo.BigSessions
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 <= 1000000
 )  AS D;

ALTER TABLE dbo.BigSessions
    ADD CONSTRAINT PK_BigSessions PRIMARY KEY(keycol);
CREATE INDEX idx_nc_app_st_et
    ON dbo.BigSessions(app, starttime, endtime);
```

I populated the table with sessions that start at random times over a period of one month and that last up to 20 minutes. I also distributed 10 different application names randomly. Now request an estimated execution plan for the original query and you will get the plan shown in Figure 3-61.



Estimated Number of Rows	981013
Estimated Row Size	23 B
Estimated Data Size	22 MB

Estimated Number of Rows	1000000
Estimated Row Size	23 B
Estimated Data Size	22 MB

SELECT

Cached plan size	24 B
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	108972
Estimated Number of Rows	10

Statement

```
SELECT app, MAX(concurrent) AS mx
FROM (SELECT app,
      (SELECT COUNT(*)
       FROM dbo.BigSessions 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.BigSessions) AS S1) AS
C
GROUP BY app;
```

Figure 3-61: Estimated execution plan for query against the BigSessions table with adequate sample data

The cost of the query is now 108,972. Trust me; you don't want to run it to see how long it really takes. Or, if you like, you can start running it and come back the next day hoping that it finished.

Now that the sample data is more realistic, you can see that the set-based solution presented in this section is slow—unlike what you might be led to believe when using inadequate sample data. In short, you can see how vital it is to put some thought into preparing good sample data. Of course, the tuning process only starts now; you might want to consider query revisions, cursor-based solutions, revisiting the model, and so on. But here I wanted to focus the discussion on bad sample data. I'll conduct a more thorough tuning discussion related to the problem at hand in *Inside T-SQL Programming.*

TABLESAMPLE

SQL Server 2005 introduces a new feature that allows you to sample data from an existing table. The tool is a clause called TABLESAMPLE that you specify after the table name in the FROM clause along with some options. Here's an example for using TABLESAMPLE to request 1000 rows from the Orders table in the Performance database:

```
SELECT *
FROM Performance.dbo.Orders TABLESAMPLE (1000 ROWS);
```

Note that if you run this query you probably won't get exactly 1000 rows. I'll explain why shortly.

You can specify TABLESAMPLE on a table-by-table basis. Following the TABLESAMPLE keyword, you can optionally specify the sampling method to use. Currently, SQL Server supports only the SYSTEM method, which is also the default if no method is specified. In the future, we might see additional algorithms. Per ANSI, the SYSTEM keyword represents an implementation-dependent sampling method. This means that you will find different algorithms implemented in different products when using the SYSTEM method. In SQL Server, the SYSTEM method implements the same sampling algorithm used to sample pages to generate statistics.

You can use either the ROWS or the PERCENT keyword to specify how many rows you would like to get back. Based on your inputs, SQL Server will calculate random values to figure out whether or not a page should be returned. Note that the decision of whether or not to read a portion of data is done at the page level. This fact, along with the fashion in which SQL Server determines whether or not to pick a page based on a random factor, means that you won't necessarily get the exact number of rows that you asked for; rather, you'll get a fairly close value. The more rows you request, the more likely you are to get a result set size close to what you requested.

Here's an example for using the TABLESAMPLE clause in a query against the Orders table, requesting 1,000 rows:

```
SET NOCOUNT ON;
USE Performance;

SELECT*
FROM dbo.Orders TABLESAMPLE SYSTEM (1000 ROWS);
```

I ran this query three times and got a different number of rows every time: 880, 1200, and 920.

An important benefit you get with the SYSTEM sampling method is that only the chosen pages (those that SQL Server picked) will be physically scanned. So even if you query a huge table, you will get the results pretty fast—as long as you specify a fairly small number of rows. As I mentioned earlier, you can also specify a percentage of rows. Here's an example requesting 0.1 percent, which is equivalent to 1,000 rows in our table:

```
SELECT*
FROM dbo.Orders TABLESAMPLE (0.1 PERCENT);
```

When you use the ROWS option, SQL Server internally first converts the specified number of rows to a percentage. Remember that you are not guaranteed to get the exact number of rows that you requested; rather, you'll get a close value that's determined by the number of pages that were picked and the number of rows on those pages (which may vary).

To make it more likely that you'll get the exact number of rows you are after, specify a higher number of rows in the TABLESAMPLE clause and use the TOP option to limit the upper bound that you will get, like so:

```
SELECT TOP(1000) *
FROM dbo.Orders TABLESAMPLE (2000 ROWS);
```

There's still a chance that you will get fewer rows than the number you requested, but you're guaranteed not to get more. By specifying a higher value in the TABLESAMPLE clause, you increase the likelihood of getting the number of rows you are after.

If you need to get repeatable results, use a clause called REPEATABLE which was designed for this purpose, providing it with the same seed in all invocations. For example, running the following query multiple times will yield the same result, provided that the data in the table has not changed:

```
SELECT *
FROM dbo.Orders TABLESAMPLE (1000 ROWS) REPEATABLE(42);
```

Note that with small tables you might not get any rows at all. For example, run the following query multiple times, requesting a single row from the Sales. StoreContact table in the AdventureWorks database:

```
SELECT *
FROM Adventure Works.Sales.StoreContact TABLESAMPLE (1 ROWS);
```

You will only occasionally get any rows back. I witnessed a very interesting discussion in a technical SQL Server forum. Someone presented such a query and wanted to know why he didn't get any rows back. Steve Kass, a friend of mine and the ingenious technical editor of these books, provided the following illuminating answer and kindly allowed me to quote him here:

> As documented in Books Online ("Limiting Results Sets by Using TABLESAMPLE"), the sampling algorithm can only return full data pages. Each page is selected or skipped with probability [desired number of rows]/[rows in table].
>
> The StoreContact table fits on 4 data pages. Three of those pages contain 179 rows, and one contains 37 rows. When you sample for 10 rows (1/75 of the table), each of the 4 pages is returned with probability 1/75 and skipped with probability 74/75. The chance that no rows are returned is about (74/75)ˆ4, or about 87%. When rows are returned, about 3/4 of the time you will see 179 rows, and about 1/4 of the time you will see 37 rows. Very rarely, you will see more rows, if two or more pages are returned, but this is very unlikely.
>
> As BOL suggests, SYSTEM sampling (which is the only choice) is not recommended for small tables. I would add that if the table fits on N data pages, you should not try to sample fewer than 1/N-th of the rows, or that you should never try to sample fewer rows than fit on at least 2 or 3 data pages.
>
> If you were to sample roughly two data pages worth of rows, say 300 rows, the chance of seeing no rows would be about 13%. The larger (more data pages) the table, the smaller the chance of seeing no rows when at least a couple of pages worth are requested. For example, if you request 300 rows from a 1,000,000-row table that fits on 10,000 data pages, only in 5% of trials would you see no rows, even though the request is for far less than 1% of the rows.
>
> By choosing the REPEATABLE option, you will get the same sample each time. For most seeds, this will be an empty sample in your case. With other seeds, it will contain 37, 179, 216, 358, 395, 537, 574, or 753 rows, depending on which pages were selected, with the larger numbers of rows returned for very few choices of seed.
>
> That said, I agree that the consequences of returning only full data pages results in very confusing behavior!

With small tables, you might want to consider other sampling methods. You don't care too much about scanning the whole table because you will consider these techniques against small tables anyway. For example, the following query will scan the whole table, but it will guarantee that you get a single random row:

```
SELECT TOP(1) *
FROM Adventure Works.Sales.StoreContact
ORDER BY NEWID();
```

Note that other database platforms, such as DB2, implement additional algorithms—for example, the Bernoulli sampling algorithm. You can implement it in SQL Server by using the following query, provided by Steve Kass:

```
SELECT *
FROM Adventure Works.Sales.StoreContact
WHERE ABS((customerid%customerid)+CHECKSUM(NEWID()))/POWER(2.,31) < 0.01
GO
```

The constant 0.01 is the desired probability (in this case, 1%) of choosing a row. The expression *customerid%customerid* was included to make the WHERE clause correlated and force its evaluation on each row of StoreContact. Without it, the value of the WHERE condition would be calculated just once, and either the entire table would be returned or no rows would be returned. Note that this technique will require a full table scan and can take a while with large tables. You can test it against our Orders table and see for yourself.

An Examination of Set-Based vs. Iterative/Procedural Approaches, and a Tuning Exercise

Thus far in the chapter, I mainly focused on index tuning for given queries. However, in large part, query tuning involves query revisions. That is, with different queries or different T-SQL code you can sometimes get substantially different plans, with widely varying costs and run times. In a perfect world, the ideal optimizer would always figure out exactly what you are trying to achieve; and for any form of query or T-SQL code that attempts to achieve the same thing, you would get the same plan—and only the best plan, of course. But alas, we're not there yet. There are still many performance improvements to gain merely from the changing way you write your code. This will be demonstrated thoroughly throughout these books. Here, I'll demonstrate a typical tuning process based on code revisions by following an example.

Note that set-based queries are typically superior to solutions based on iterative/procedural logic—such as ones using cursors, loops, and the like. Besides the fact that set-based solutions usually require much less code, they also usually involve less overhead than cursors. There's a lot of overhead incurred with the record-by-record manipulation of cursors. You can make simple benchmarks to observe the performance differences. Run a query that simply selects all rows from a big table, discarding the results in the graphical tool so that the time it takes to display the output won't be taken into consideration. Also run cursor code that simply scans all table rows one at a time. Even if you use the fastest available cursor—FAST_FORWARD (forward only, read only)—you will find that the set-based query runs dozens of times faster. Besides the overhead involved with a cursor, there's also an issue with the execution plans. When using a cursor, you apply a very rigid physical approach to accessing the data, because your code focuses a lot on how to achieve the result. A set-based query, on the other hand, focuses logically on *what* you want to achieve rather than *how* to achieve it. Typically, set-based queries leave the optimizer with much more room for maneuvering and leeway to do what it is good at—optimization.

That's the rule of thumb. However, I'm typically very careful with adopting rules of thumb, especially with regard to query tuning—because optimization is such a dynamic world, and there are always exceptions. In fact, as far as query tuning is concerned, my main rule of thumb is to be careful about adopting rules of thumb.

You will encounter cases where it is very hard to beat cursor code, and you need to be able to identify them; but these cases are the minority. I'll discuss the subject at length in Chapter 3, "Cursors," of *Inside T-SQL Programming.*

To demonstrate a tuning process based on code revisions, I'll use our Orders and Shippers tables. The request is to return shippers that used to be active, but do not have any activity as of January 1, 2001. That is, a qualifying shipper is one for whom you cannot find an order on or after January 1, 2001. You don't care about shippers who have made no orders at all.

Before you start working, remove all indexes from the Orders table, and make sure that you have only the clustered index defined on the *orderdate* column and the primary key (nonclustered) defined on the *orderid* column.

If you rerun the code in Listing 3-1, make sure that for the Orders table, you keep only the following index and primary key definitions:

```
SET NOCOUNT ON;
USE Performance;
CREATE CLUSTERED INDEX idx_cl_od ON dbo.Orders(orderdate);
ALTER TABLE dbo.Orders ADD
  CONSTRAINT PK_Orders PRIMARY KEY NONCLUSTERED(orderid);
```

Next, run the following code to add a few shippers to the Shippers table and a few orders to the Orders table:

```
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('B','Shipper_B');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('D','Shipper_D');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('F','Shipper_F');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('H','Shipper_H');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('X','Shipper_X');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('Y','Shipper_Y');
INSERT INTO dbo.Shippers(shipperid, shippername) VALUES('Z','Shipper_Z');

INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  VALUES(1000001, 'C0000000001', 1, 'B', '20000101');
INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  VALUES(1000002, 'C0000000001', 1, 'D', '20000101');
INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  VALUES(1000003, 'C0000000001', 1, 'F', '20000101');
INSERT INTO dbo.Orders(orderid, custid, empid, shipperid, orderdate)
  VALUES(1000004, 'C0000000001', 1, 'H', '20000101');
```

You're supposed to get the shipper IDs B, D, F, and H in the result. These are the only shippers that were active at some point, but not as of January 1, 2001.

In terms of index tuning, it's sometimes hard to figure out what the optimal indexes are without having an existing query to tune. But in our case, index tuning is rather simple and possible without having the solution code first. Obviously, you will want to search for an *orderdate* value for each *shipperid*, so naturally the optimal index would be a nonclustered covering index defined with *shipperid* and *orderdate* as the key columns, in that order:

```
CREATE NONCLUSTERED INDEX idx_nc_sid_od
  ON dbo.Orders(shipperid, orderdate);
```

I suggest that at this point you try to come up with the best performing solution that you can; then compare it with the solutions that I will demonstrate.

As the first solution, I'll start with the cursor-based code shown in Listing 3-9.

Listing 3-9: Cursor solution

```
DECLARE
  @sid     AS VARCHAR(5),
  @od      AS DATETIME,
  @prevsid AS VARCHAR(5),
  @prevod  AS DATETIME;

DECLARE ShipOrdersCursor CURSOR FAST_FORWARD FOR
  SELECT shipperid, orderdate
  FROM dbo.Orders
  ORDER BY shipperid, orderdate;

OPEN ShipOrdersCursor;
FETCH NEXT FROM ShipOrdersCursor INTO @sid, @od;

SELECT @prevsid = @sid, @prevod = @od;

WHILE @@fetch_status = 0
BEGIN
  IF @prevsid <> @sid AND @prevod < '20010101'PRINT @prevsid;
  SELECT @prevsid = @sid, @prevod = @od;
  FETCH NEXT FROM ShipOrdersCursor INTO @sid, @od;
END

IF @prevod < '20010101' PRINT @prevsid;

CLOSE ShipOrdersCursor;

DEALLOCATE ShipOrdersCursor;
```

This code implements a straightforward data-aggregation algorithm based on sorting. The cursor is defined on a query that sorts the data by *shipperid* and *orderdate*, and it scans the records in a forward-only, read-only manner—the fastest scan you can get with a cursor. For each shipper, the code inspects the last row found—which happens to hold the maximum *orderdate* for that shipper—and if that date is earlier than ‘20010101’, the code emits the *shipperid* value. This code ran on my machine for 27 seconds. Imagine the run time in a larger Orders table that contains millions of rows.

The next solution (call it *set-based solution 1*) is a natural GROUP BY query that many programmers would come up with:

```
SELECT shipperid
FROM dbo.Orders
GROUP BY shipperid
HAVING MAX(orderdate) < '20010101';
```

You just say what you want, rather than spending most of your code describing how to get it. The query groups the data by *shipperid*, and it returns only shippers with a maximum *orderdate* that is earlier than ‘20010101’.

This query ran for just under one second in my machine. The optimizer produced the execution plan shown in Figure 3-62 for this query.

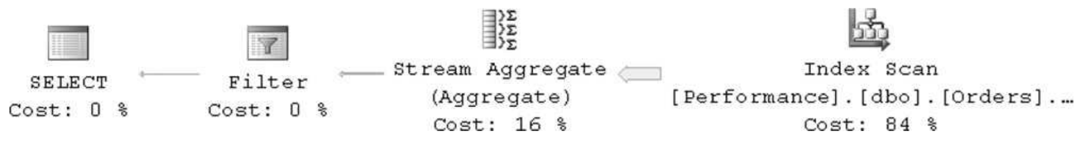


Figure 3-62: Execution plan for set-based solution 1

The plan shows that our covering index was fully scanned in order. The maximum *orderdate* was isolated for each *shipperid* by the Stream Aggregate operator. Then the filter operator filtered only shippers for whom the maximum *orderdate* was before ‘20010101’.

Here are the vital performance measures I got for this query:

- Logical reads: 2730
- CPU time: 670 ms
- Elapsed time: 732 ms

Note that you might get slightly different performance measures. At this point, you need to ask yourself if you're happy with the result, and if you're not, whether there's potential for optimization at all.

Of course, this solution is a big improvement over the cursor-based one, both in terms of performance and code readability and maintenance. However, a run time of close to one second for such a query might not be satisfactory. Keep in mind that an Orders tables in some production environments can contain far more than one million rows.

If you determine that you want to tune the solution further, you now need to figure out whether there's potential for optimization. Remember that in the execution plan for the last query, the leaf level of the index was fully scanned to obtain the latest *orderdate* for each shipper. That scan required 2730 page reads. Our Shippers table contains 12 shippers. Your gut feelings should tell you that there must be a way to obtain the data with much fewer reads. In our index, the rows are sorted by *shipperid* and *orderdate.* This means that there are groups of rows—a group for each *shipperid*—where the last row in each group contains the latest *orderdate* that you want to inspect. Alas, the optimizer currently doesn't have the logic within it to "zigzag" between the levels of the index, jumping from one shipper's latest *orderdate* to the next. If it had, the query would have incurred substantially less I/O. By the way, such zigzagging logic can be beneficial for other types of requests—for example, requests involving filters on a non-first index column and others as well. But I won't digress.

Of course, if you request the latest *orderdate* for a particular shipper, the optimizer can use a seek directly to the last shipper's row in the index. Such a seek would cost 3 reads in our case. Then the optimizer can apply a TOP operator going one step backwards, returning the desired value—the latest *orderdate* for the given shipper—to a Stream Aggregate operator.

The following query demonstrates acquiring the latest *orderdate* for a particular shipper, producing the execution plan shown in Figure 3-63:

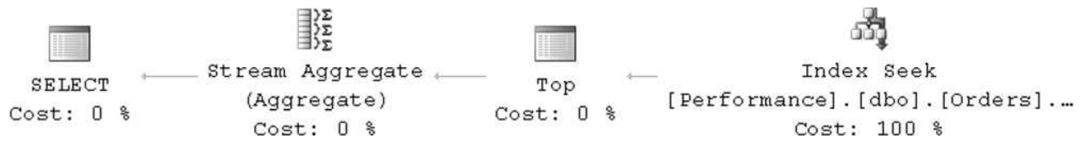


Figure 3-63: Execution plan for a query handling a particular shipper

```
SELECT MAX(orderdate) FROM dbo.Orders WHERE shipperid = 'A';
```

This plan incurs only 5 logical reads. Now, if you do the math for 12 shippers, you will realize that you can potentially obtain the desired result with substantially less I/O than 2730 reads.

Of course, you could scan the Shippers rows with a cursor, and then invoke such a query for each shipper; but it would be counterproductive and a bit ironic to beat a cursor solution with a set-based solution, that you then beat with another cursor.

Realizing that what you're after is invoking a seek operation for each shipper, you might come up with the following attempt (call it *set-based solution 2*):

```
SELECT shipperid
```

```
FROM (SELECT shipperid,
       (SELECT MAX(orderdate)
        FROM dbo.Orders AS O
        WHERE O.shipperid = S.shipperid) AS maxod
     FROM dbo.Shippers AS S) AS D
WHERE maxod < '20010101';
```

You query the Shippers table, and for each shipper, a subquery acquires the latest *orderdate* value (aliased as *maxod*). The outer query then filters only shippers with a *maxod* value that is earlier than ‘20010101’. Shippers who have never placed an order will be filtered out; for those, the subquery will yield a NULL, and a NULL compared to any value yields UNKNOWN, which in turn is filtered out.

But strangely enough, you get the plan shown in Figure 3-64, which looks surprisingly similar to the previous one.

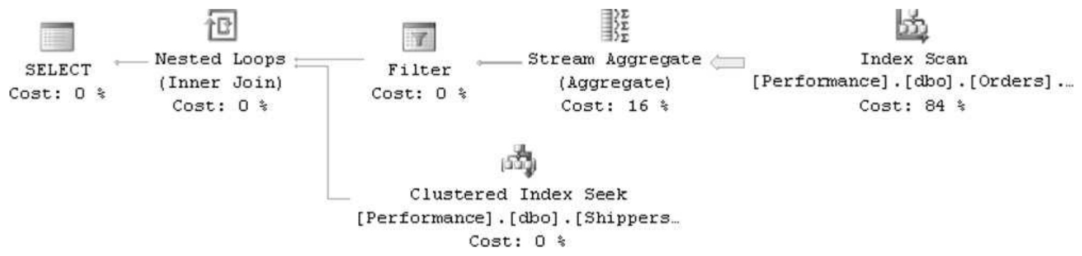


Figure 3-64: Execution plan for set-based solution 2

And the performance measures are also similar to the previous ones. This query also incurred 2730 logical reads against the Orders table, and ran for close to one second on my machine. So what happened here?

It seems that the optimizer got "too sophisticated" this time. One guess is that it realized that for shippers who have made no orders, the subquery would return a NULL and they would be filtered out in the outer query's filter—meaning that only shippers that do appear in the Orders table are of interest. Thus, it internally "simplified" the query to one that is similar to our set-based solution 1, hence the similar plans.

The situation seems to be evolving into a battle of wits with the optimizer—not a battle to the death, of course; there won't be any iocane powder involved here, just I/O. The optimizer pulls a trick on you; now pull your best trick. You can make the optimizer believe that shippers with no orders might also be of interest, by substituting a NULL value returned from the subquery with a constant; you can do so by using the COALESCE function. Of course, you'd use a constant that will never allow the filter expression to yield TRUE, as you're not interested in shippers that have no associated orders.

You issue the following query (call it *set-based solution 3*), close your eyes, and hope for the best:

```
SELECT shipperid
FROM (SELECT shipperid,
        (SELECT MAX(orderdate)
         FROM dbo.Orders AS O
         WHERE O.shipperid = S.shipperid) AS maxod
      FROM dbo.Shippers AS S)AS D
WHERE COALESCE(maxod, '99991231') < '20010101';
```

And when you open your eyes, voilà! You see the plan you wished for, as shown in Figure 3-65.

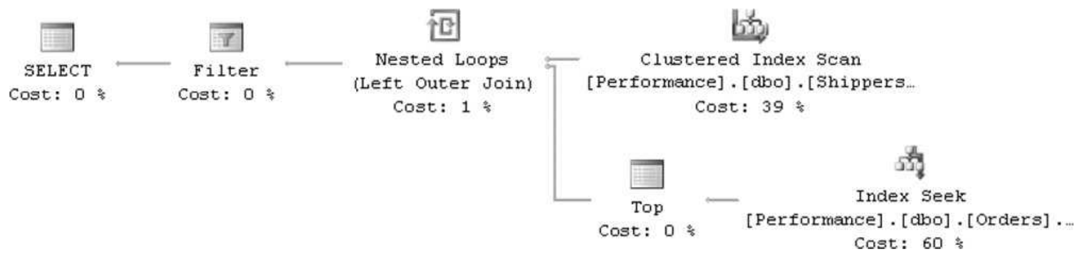


Figure 3-65: Execution plan for set-based solution 3

The Shippers table is scanned, and for each of the 12 shippers, a Nested Loops operator invokes a similar activity to the one you got when invoking a query for a particular shipper. This plan incurs only 60 logical reads. The net CPU time is not even measurable with STATISTICS TIME (shows up as 0), and I got 26 milliseconds of elapsed time.

Once you get over the excitement of outwitting the optimizer, you start facing some troubling facts. Such a query is not very natural, and it's not the most intuitive for programmers to grasp. It uses a very artificial technique, with the sole purpose of overcoming an optimizer issue—which is similar to using a hint, but with tricky logic applied. Some other programmer who needed to maintain your code in the future might stare at the query for awhile and then ask, "What on earth was the programmer who wrote this query thinking? I can do this much more simply." The programmer would then revise the code back to set-based solution 1. Of course, you can add comments documenting the technique and the reasoning behind it. But humor aside, the point is that queries should be simple, natural, and intuitive for people to read and understand—of course, as much as that's humanly possible. A complex solution, as fast as it might be, is not a good solution. Bear in mind that our case is a simplistic scenario used for illustration purposes. Typical production queries involve several joins, more filters, and more logic. Adding such "tricks" would only add to the complexity of the queries.

In short, you shouldn't be satisfied with this solution, and you should keep looking for other alternatives.

And if you look hard enough, you will find this one (call it *set-based solution 4*):

```
SELECT shipperid
FROM dbo.Shippers AS S
WHERE NOT EXISTS
  (SELECT *FROM dbo.Orders AS O
   WHERE O.shipperid = S.shipperid
     AND O.orderdate >= '20010101')
   AND EXISTS
   (SELECT *FROM dbo.Orders AS O
    WHERE O.shipperid = S.shipperid);
```

This solution is natural. You query the Shippers table and filter shippers for whom you cannot find an order on or past ‘20010101’ and for whom you can find at least one order.

You get the plan shown in Figure 3-66.

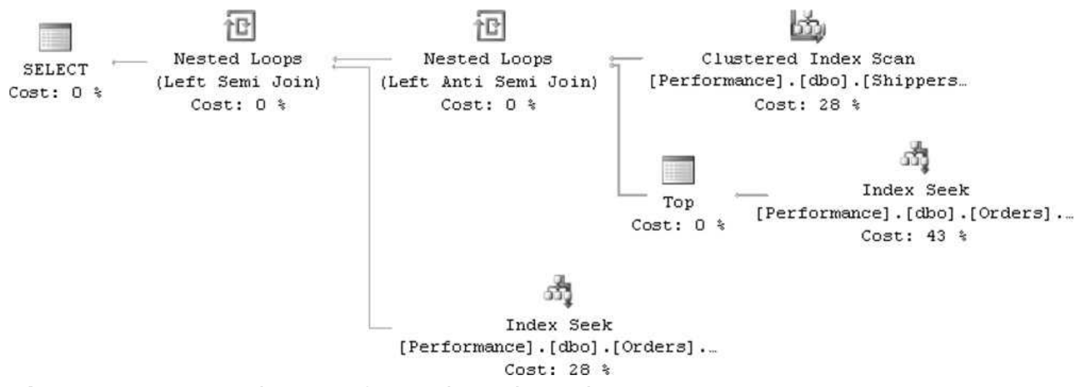


Figure 3-66: Execution plan for set-based solution 4

The Shippers table is scanned, yielding 12 rows. For each shipper, a Nested Loops operator invokes a seek against our covering index to check whether an *orderdate* of ‘20010101’ or later exists for the shipper. If the answer is yes, another seek operation is invoked against the index to check whether an order exists at all. The I/O cost against the Orders table is 78 reads— slightly higher than the previous solution. However, in terms of simplicity and naturalness this solution wins big time! Therefore, I would stick to it.

As you probably realize, index tuning alone is not enough; there's much you can do with the way you write your queries. Being a "Matrix" fan, I'd like to believe that it's not the spoon that bends; it's only your mind.

Additional Resources

As promised, here is a list of resources where you can find more information about the subjects discussed in this chapter:

- Later in this book, you will find coverage of the querying techniques used in this chapter, including subqueries, table expressions and ranking functions, joins, aggregations, TOP, data modification, recursive queries, and others.
- The *Inside Microsoft SQL Server 2005: T-SQL Programming* book focuses on areas related to T-SQL programming,

some of which were touched on in this chapter, including datatypes, temporary tables, tempdb, row versioning, cursors, dynamic execution, T-SQL and CLR routines, compilations/recompilations, concurrency, and others.

- The books *Inside Microsoft SQL Server 2005: The Storage Engine* and *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*, both written by Kalen Delaney (Microsoft Press, 2006 and 2007), cover various internals-related subjects in great depth, including the storage engine, indexes, transaction log architecture, the query optimizer, concurrency, dynamic management objects, tracing and Profiler, and many others as well.
- You can find more information about wait types and analyzing wait statistics at the following links:
  - ❑ "Opening Microsoft's Performance-Tuning Toolbox" by Tom Davidson (http://www.windowsitpro.com/Article/ArticleID/40925/ 40925.html)
  - ❑ Gert Drapers' Web site: SQLDev. Net (http://www.sqldev.net/)
- You can find information about balanced trees at the following sources:
  - ❑http://www.nist.gov/dads/HTML/balancedtree.html
  - ❑*The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)* by Donald E. Knuth (Addison-Wesley Professional, 1998)
- The following white papers elaborate on subjects touched on in this chapter:
  - ❑ "Troubleshooting Performance Problems in SQL Server 2005," written by various authors (http://www.microsoft.com/technet/prodtechnol/sql/2005/ tsprfprb.mspx)
  - ❑ "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005" by Arun Marathe (http://www.microsoft.com/technet/prodtechnol/sql/2005/ recomp.mspx)
  - ❑ "Statistics Used by the Query Optimizer in Microsoft SQL Server 2005" by Eric N. Hanson (http://www.microsoft.com/technet/prodtechnol/sql/2005/ qrystats.mspx)
  - ❑ "Forcing Query Plans with SQL Server 2005" by Burzin A. Patel (*http://www.download.microsoft.com/download/1/3/4/134644FD- 05AD-4EE8-8B5A-0AED1C18A31E/Forcing_Query_Plans.doc*)
  - ❑ "Improving Performance with SQL Server 2005 Indexed Views" by Eric Hanson (http://www.microsoft.com/technet/prodtechnol/sql/2005/ ipsql05iv.mspx)
  - ❑ "Database Concurrency and Row Level Versioning in SQL Server 2005" by Kalen Delaney and Fernando Guerrero (http://www.microsoft.com/technet/prodtechnol/sql/2005/ cncrrncy.mspx)
- You can find a series of articles I wrote covering table and index partitioning at the *SQL Server Magazine* Web site (http://www.sqlmag.com). Look up the following instant document IDs: 45153, 45533, 45877, and 46207.
- You can find all resources related to this book at http://www.insidetsql.com.

Conclusion

This chapter covered a tuning methodology, index tuning, the importance of sample data, and query tuning by query revisions. There's so much involved in tuning that knowledge of the product's architecture and internals plays a big role in doing it well. But knowledge is not enough. I hope this chapter gave you the tools and guidance that will allow you to put your knowledge into action as you progress in these books—and, of course, in your production environments.