Inside Microsoft SQL Server 2005: T-SQL Querying

by Itzik Ben-Gan, Lubor Kollar and Dejan Sarka

Chapter 7: Top and Apply

This chapter covers two query elements that might seem unrelated to each other. One element is the TOP option, which allows you to limit the number of rows affected by a query. The other is the new APPLY table operator, which allows you to apply a table expression to each row of another table expression—basically creating a correlated join. I decided to cover both in the same chapter because I find that quite often you can use them together to solve querying problems.

I'll first describe the fundamentals of TOP and APPLY, and then follow with solutions to common problems using these elements.

SELECT TOP

In a SELECT query or table expression, TOP is used with an ORDER BY clause to limit the result to rows that come first in the ORDER BY ordering. You can specify the quantity of rows you want in one of two ways: as an exact number of rows, from TOP(0) to TOP(9223372036854775807) (the largest BIGINT value), or as a percentage of rows, from TOP(0E0) PERCENT to TOP(100E0) PERCENT, using a FLOAT value. In Microsoft SQL Server 2000, you could only use a constant to specify the limit. SQL Server 2005 supports any self-contained expression, not just constants, with TOP.

To make it clear which rows are the "top" rows affected by a TOP query, you must indicate an ordering of the rows. Just as you can't tell top from bottom unless you know which way is up, you won't know which rows TOP affects unless you specify an ORDER BY clause. You should think of TOP and ORDER BY together as a logical filter rather than a sorting mechanism. That's why a query with both a TOP clause and an ORDER BY clause returns a well-defined table and is allowed in table expressions. In a query without TOP, an ORDER BY clause has a different purpose—it simply specifies the order in which results are returned. Using ORDER BY without TOP is not allowed in table expressions.

> **Note** Interestingly, you can specify the TOP option in a query without an ORDER BY clause, but the logical meaning of TOP in such a query is not completely defined. I'll explain this aspect of TOP shortly.

Let's start with a basic example. The following query returns the three most recent orders, producing the output shown in Table 7-1:

```
USE Northwind;
SELECT TOP(3)OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderID DESC;
```

Table 7-1: Three Most Recent Orders

OrderID	CustomerID	OrderDate
11077	RATTC	1998-05-06 00:00:00.000
11076	BONAP	1998-05-06 00:00:00.000
11075	RICSU	1998-05-06 00:00:00.000

Sorting first by *OrderDate* DESC guarantees that you will get the most recent orders. Because *OrderDate* is not unique, I added *OrderID* DESC to the ORDER BY list as a tiebreaker. Among orders with the same *OrderDate*, the tiebreaker will give precedence to orders with higher *OrderID* values.

> **Note** Notice the usage of parentheses here for the input expression to the TOP option. Because SQL Server 2005 supports any self-contained expression as input, the expression must reside within parentheses. For backward-compatibility reasons, SQL Server 2005 still supports SELECT TOP queries that use a constant without parentheses. However, it's good practice to put TOP constants in parentheses to conform to the new requirements.

As an example of the PERCENT option, the following query returns the most recent one percent of orders, generating the output shown in Table 7-2:

```
SELECT TOP(1)PERCENT OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderID DESC;
```

Table 7-2: Most Recent One Percent of Orders

OrderID	CustomerID	OrderDate
11077	RATTC	1998-05-06 00:00:00.000
11076	BONAP	1998-05-06 00:00:00.000
11075	RICSU	1998-05-06 00:00:00.000
11074	SIMOB	1998-05-06 00:00:00.000
11073	PERIC	1998-05-05 00:00:00.000
11072	ERNSH	1998-05-05 00:00:00.000
11071	LILAS	1998-05-05 00:00:00.000
11070	LEHMS	1998-05-05 00:00:00.000
11069	TORTU	1998-05-04 00:00:00.000

The Orders table has 830 rows, and one percent of 830 is 8.3. Because only whole rows can be returned and 8.3 were requested, the actual number of rows returned is 9. When TOP … PERCENT is used, and the specified percent includes a fractional row, the exact number of rows requested is rounded up.

TOP and Determinism

As I mentioned earlier, a TOP query doesn't require an ORDER BY clause. However, such a query is nondeterministic. That is, running the same query twice against the same data might yield different result sets, and both would be correct. The following query returns three orders, with no rule governing which three are returned:

```
SELECT TOP(3) OrderID, CustomerID, OrderDate
FROM dbo.Orders;
```

When I ran this query, I got the output shown in Table 7-3, but you might get a different output. SQL Server will return the first three rows it happened to access first.

Table 7-3: Result of TOP Query with No ORDER BY

OrderID	CustomerID	OrderDate
10248	VINET	1996-07-04 00:00:00.000
10249	TOMSP	1996-07-05 00:00:00.000
10250	HANAR	1996-07-08 00:00:00.000

Note I can think of only two good reasons to use SELECT TOP without ORDER BY, and I don't recommend it otherwise. One reason to use SELECT TOP is to serve as a quick reminder of the structure or column names of a table, or to find out if the table contains any data at all. The other reason to use SELECT TOP—specifically SELECT TOP(0)—is to create an empty table with the same structure as another table or query. In this case, you can use SELECT TOP(0) <column list> INTO <table name> FROM …. Obviously, you don't need an ORDER BY clause to indicate "which zero rows" you want to select!

A TOP query can be nondeterministic even when an ORDER BY clause is specified, if the ORDER BY list is nonunique. For example, the following query returns the first three orders in order of increasing *CustomerID*, generating the output shown in Table 7-4:

```
SELECT TOP(3) OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY CustomerID;
```

Table 7-4: Result of TOP Query with Nonunique ORDER BY

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

You are guaranteed to get the orders with the lowest *CustomerID* values. However, because the *CustomerID* column is not unique, you cannot guarantee which rows among the ones with the same *CustomerID* values will be returned in case of ties. Again, you will get the ones that SQL Server happened to access first. One way to guarantee determinism is to add a tiebreaker that makes the ORDER BY list unique—for example, the primary key:

```
SELECT TOP(3) OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY CustomerID, OrderID;
```

Another way to guarantee determinism is to use the WITH TIES option. When you use WITH TIES, the query will generate a result set including any additional rows that have the same values in the sort column or columns as the last row returned. For example, the following query specifies TOP(3), yet it returns the seven rows shown in Table 7-5. Four additional orders are returned because they have the same *CustomerID* value (ALFKI) as the third row:

```
SELECT TOP(3) WITH TIES OrderID, CustomerID, OrderDate
FROM dbo.Orders
ORDER BY CustomerID;
```

Table 7-5: Result of TOP Query Using the WITH TIES Option

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

Note Some applications must guarantee determinism. For example, if you're using the TOP option to implement paging, you don't want the same row to end up on two successive pages just because the query was nondeterministic. Remember that you can always add the primary key as a tiebreaker to guarantee determinism in case the ORDER BY list is not unique.

TOP and Input Expressions

As the input to TOP, SQL Server 2005 supports any self-contained expression yielding a scalar result. An expression that is independent of the outer query can be used—a variable, an arithmetic expression, or even the result of a subquery. For example, the following query returns the @*n* most recent orders, where @*n* is a variable:

```
DECLARE @n AS INT;
SET@n = 2;

SELECT TOP(@n)OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER  BY OrderDate DESC, OrderIDDESC;
```

The following query shows the use of a subquery as the input to TOP. As always, the input to TOP specifies the number of rows the query returns—for this example, the number of rows returned is the monthly average number of orders. The ORDER BY clause in this example specifies that the rows returned are the most recent ones, where *OrderID* is the tiebreaker (higher ID wins):

```
SELECT  TOP(SELECT COUNT(*)/(DATEDIFF(month,
             MIN(OrderDate), MAX(OrderDate))+1)
           FROM dbo.Orders)
    OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY OrderDate DESC, OrderIDDESC;
```

The average number of monthly orders is the count of orders divided by one more than the difference in months between the maximum and minimum order dates. Because there are 830 orders in the table that were placed during a period of 23 months, the output has the most recent 36 orders.

TOP and Modifications

SQL Server 2005 provides a TOP option for data modification statements (INSERT, UPDATE, and DELETE).

Note Before SQL Server 2005, the SET ROWCOUNT option provided the same capability as some of TOP's new features. SET ROWCOUNT accepted a variable as input, and it affected both data modification statements and SELECT statements. Microsoft no longer recommends SET ROWCOUNT as a way to affect INSERT, UPDATE, and DELETE statements—in fact, in Katmai, the next release of SQL Server, SET ROWCOUNT will not affect data modification statements at all. Use TOP to limit the number of rows affected by data modification statements.

SQL Server 2005 introduces support for the TOP option with modification statements, allowing you to limit the number or percentage of affected rows. A TOP specification can follow the keyword DELETE, UPDATE, or INSERT.

An ORDER BY clause is not supported with modification statements, even when using TOP, so none of them can rely on ordering. SQL Server will simply affect the specified number of rows that it happens to access first.

In the following statement, SQL Server does not guarantee which rows will be inserted from the source table:

```
INSERT TOP(10)INTO target_table
    SELECT col1, col2, col3
    FROM source_table;
```

Note Although you cannot use ORDER BY with INSERT TOP, you can guarantee which rows will be inserted if you specify TOP and ORDER BY in the SELECT statement, like so:

```
INSERT INTO target_table
    SELECT TOP(10)col1, col2,  col3
    FROM source_table
    ORDER BY col1;
```

An INSERT TOP is handy when you want to load a subset of rows from a large table or result set into a target table and you don't care which subset will be chosen; instead, you care only about the number of rows.

Note Although ORDER BY cannot be used with UPDATE TOP and DELETE TOP, you can overcome the limitation by creating a CTE from a SELECT TOP query that has an ORDER BY clause and then issue your UPDATE or DELETE against the CTE:

```
WITH CTE_DEL AS
(
    SELECT TOP(10)*  FROM some_table ORDERBY col1
)
DELETE FROM CTE_DEL;

WITH CTE_UPD AS
(
    SELECT TOP(10)*  FROM some_table ORDERBY col1
)
UPDATE CTE_UPD SET col2=  col2+  1;
```

One such situation is when you need to insert or modify large volumes of data and, for practical reasons, you split it into batches, modifying one subset of the data at a time. For example, purging historic data might involve deleting millions of rows of data. Unless the target table is partitioned and you can simply drop a partition, the purging process requires a DELETE statement. Deleting such a large set of rows in a single transaction has several drawbacks. A DELETE statement is fully logged, and it will require enough space in the transaction log to accommodate the whole transaction. During the delete operation, which could take a long time, no part of the log from the oldest open transaction up to the current point can be overwritten. Furthermore, if the transaction breaks in the middle for some reason, all the activity that took place to that point will be rolled back, and this will take a while. Finally, when very many rows are deleted at once, SQL Server might escalate the individual locks held on the deleted rows to an exclusive table lock, preventing both read and write access to the target table until the DELETE is completed.

It makes sense to break the single large DELETE transaction into several smaller ones—small enough not to cause lock escalation (typically, a few thousand rows per transaction), and allowing recycling of the transaction log. You can easily verify that the number you chose doesn't cause lock escalation by testing a DELETE with the TOP option while monitoring Lock Escalation events with Profiler. Splitting the large DELETE will also allow overwriting the inactive section of the log that was already backed up.

To demonstrate purging data in multiple transactions, run the following code, which creates the LargeOrders table and populates it with sample data:

```
IF OBJECT_ID('dbo.LargeOrders') IS NOT NULL
    DROP TABLE dbo.LargeOrders;
GO
SELECT IDENTITY(int,  1,  1)AS OrderID,
    O1.CustomerID,  O1.EmployeeID,  O1.OrderDate,  O1.RequiredDate,
    O1.ShippedDate,  O1.ShipVia, O1.Freight,  O1.ShipName, O1.ShipAddress,
    O1.ShipCity, O1.ShipRegion, O1.ShipPostalCode,  O1.ShipCountry
INTO dbo.LargeOrders
FROM dbo.Ordersv AS O1, dbo.Orders AS O2;

CREATE UNIQUE CLUSTERED INDEX idx_od_oid
    ON dbo.LargeOrders(OrderDate,  OrderID);
```

In versions prior to SQL Server 2005, use the SET ROWCOUNT option to split a large DELETE, as the following solution shows:

```
SET ROWCOUNT 5000;
WHILE 1 = 1
BEGIN
    DELETEF ROM dbo.LargeOrders
    WHERE OrderDate  <'19970101';

    IF @@rowcount < 5000 BREAK;
END
SET ROWCOUNT 0;
```

The code sets the ROWCOUNT option to 5000, limiting the number of rows affected by any DML to 5000. An endless loop attempts to delete 5000 rows in each iteration, where each 5000-row deletion resides in a separate transaction. The loop breaks as soon as the last batch is handled (that is, when the number of affected rows is less than 5000).

In SQL Server 2005, you don't need the SET ROWCOUNT option anymore. Simply specify DELETE TOP(5000):

```
WHILE 1 =  1
BEGIN
    DELETE TOP(5000)FROM dbo.Large Orders
    WHERE OrderDate> '19970101';

    IF @@rowcount < 5000 BREAK;
END
```

In a similar manner, you can split large updates into batches. For example, say you need to change the *CustomerID* OLDWO to ABCDE wherever it appears in the LargeOrders table. Here's the solution you would use in SQL Server 2000 when relying on the SET ROWCOUNT option:

```
SET ROWCOUNT 5000;
WHILE 1 =  1
BEGIN
    UPDATE dbo.Large Orders
       SET CustomerID = N'ABCDE'
    WHERE CustomerID = N'OLDWO';

    IF @@rowcount < 5000 BREAK;
END
SETR OWCOUNT 0;
```

And here's the solution in SQL Server 2005 using UPDATE TOP:

```
WHILE 1 =  1
BEGIN
    UPDATE TOP(5000) dbo.LargeOrders
       SET CustomerID = N'ABCDE'
    WHERE CustomerID = N'OLDWO';

    IF @@rowcount < 5000 BREAK;
END
```

When you're done experimenting with the batch modifications, drop the LargeOrders table:

```
IF OBJECT_ID('dbo.LargeOrders')IS NOTNULL
    DROP TABLE dbo.LargeOrders;
```

APPLY

The new APPLY table operator applies the right-hand table expression to every row of the left-hand table expression. Unlike a join, where there's no importance to the order in which each of the table expressions is evaluated, APPLY must logically evaluate the left table expression first. This logical evaluation order of the inputs allows the right table expression to be correlated with the left one—something that was not possible prior to SQL Server 2005. The concept can probably be made clearer with an example.

Run the following code to create an inline table-valued function called *fn_top_products*:

```
IF OBJECT_ID('dbo.fn_top_products') IS NOT NULL
    DROP FUNCTION dbo.fn_top_products;
GO
CREATE FUNCTION dbo.fn_top_products
    (@supid ASINT,  @catid INT,@n AS INT)
    RETURNS TABLE
AS
RETURN
    SELECT TOP(@n) WITH TIES ProductID, ProductName, UnitPrice
    FROM dbo.Products
    WHERES upplierID =   @supid
        AND CategoryID = @catid
    ORDERBY UnitPrice DESC;
GO
```

The function accepts three inputs: a supplier ID *(@supid*), a category ID (*@catid*), and a requested number of products (*@n*). The function returns the requested number of products of the given category, supplied by the given supplier, with the highest unit prices. The query uses the TOP option WITH TIES to ensure a deterministic result set by including all products that have the same unit price as the least expensive product returned.

The following query uses the APPLY operator in conjunction with *fn_top_products* to return, for each supplier, the two most expensive beverages. The Category ID for beverages is 1, so 1 is supplied for the parameter *@catid*. This query generates the output shown in Table 7-6:

```
SELECT S.SupplierID,  Company Name, ProductID,  ProductName, UnitPrice
FROM dbo.Suppliers AS S
    CROSS APPLY dbo.fn_top_products(S.SupplierID, 1, 2) AS P;
```

Table 7-6: Two Most Expensive Beverages for Each Supplier

SupplierID	CompanyName	ProductID	ProductName	UnitPrice
18	Aux joyeux ecclésiastiques	38	Côte de Blaye	263.50
18	Aux joyeux ecclésiastiques	39	Chartreuse verte	18.00
16	Bigfoot Breweries	35	Steeleye Stout	18.00
16	Bigfoot Breweries	67	Laughing Lumberjack Lager	14.00
16	Bigfoot Breweries	34	Sasquatch Ale	14.00
1	Exotic Liquids	2	Chang	19.00
1	Exotic Liquids	1	Chai	18.00
23	Karkki Oy	76	Lakkalikööri	18.00
20	Leka Trading	43	Ipoh Coffee	46.00
7	Pavlova, Ltd.	70	Outback Lager	15.00
12	Plutzer Lebensmittel-groβmärkte AG	75	Rhönbräu Klosterbier	7.75
10	Refrescos Americanas LTDA	24	Guaraná Fantástica	4.50

Specifying CROSS with the APPLY operator means that there will be nothing in the result set for a row in the left table expression for which the right table expression *dbo.fn_top_ products(S.SupplierID, 1, 2)* is empty. Such is the case here, for example, for suppliers that don't supply beverages. To include results for those suppliers as well, use the OUTER keyword instead of CROSS, as the following query shows:

```
SELECT S.SupplierID, CompanyName, ProductID, ProductName, UnitPrice
FROM dbo.Suppliers AS S
    OUTER APPLY dbo.fn_top_products(S.SupplierID,  1, 2) AS P;
```

This query returns 33 rows. The result set with OUTER APPLY includes left rows for which the right table expression yielded an empty set, and for these rows the right table expression's attributes are NULL.

There's a nice side-effect that resulted from the technology added to SQL Server's engine to support the APPLY operator. Now you are allowed to pass a column reference parameter from an outer query to a table-valued function. As an example of this capability, the following query returns, for each supplier, the lower of the two most expensive beverage prices (assuming there are at least two), generating the output shown in Table 7-7:

```
SELECT S.SupplierID, CompanyName,
    (SELECT MIN(UnitPrice)
      FROM dbo.fn_top_products(S.SupplierID, 1, 2) AS P) AS Price
FROM dbo.Suppliers AS S;
```

Table 7-7: The Lower of the Two Most Expensive Beverages per Supplier

SupplierID	CompanyName	Price
18	Aux joyeux ecclésiastiques	18.00
16	Bigfoot Breweries	14.00
5	Cooperativa de Quesos 'LasCabras'	NULL
27	Escargots Nouveaux	NULL
1	Exotic Liquids	18.00
29	Forêts d'érables	NULL
14	Formaggi Fortini s.r.l.	NULL
28	Gai pâturage	NULL
24	G'day,	Mate
3	Grandma Kelly's Homestead	NULL
11	Heli Süβwaren GmbH & Co. KG	NULL
23	Karkki Oy	18.00
20	Leka Trading	46.00
21	Lyngbysild	NULL
25	Ma Maison	NULL
6	Mayumi's	NULL
19	New England Seafood Cannery	NULL
2	New Orleans Cajun Delights	NULL
13	Nord-Ost-Fisch Handelsgesellschaft mbH	NULL
15	Norske Meierier	NULL
26	Pasta Buttini s.r.l.	NULL
7	Pavlova, Ltd.	15.00
9	PB Knäckebröd AB	NULL
12	Plutzer Lebensmittelgroβmärkte AG	7.75
10	Refrescos Americanas LTDA	4.50
8	Specialty Biscuits, Ltd.	NULL
17	Svensk Sjöföda AB	NULL
4	Tokyo Traders	NULL
22	Zaanse Snoepfabriek	NULL

Solutions to Common Problems Using TOP and APPLY

Now that the fundamentals of TOP and APPLY have been covered, I'll present common problems and solutions that use TOP and APPLY.

TOP *n* for Each Group

In Chapter 4 and Chapter 6, I discussed a problem involving tiebreakers where you were asked to return the most recent order for each employee. This problem is actually a special case of a more generic problem where you are after the top *n* rows for each group—for example, returning the three most recent orders for each employee. Again, orders with higher *OrderDate* values have precedence, but you need to introduce a tiebreaker to determine precedence in case of ties. Here I'll use the maximum *OrderID* as the tiebreaker. I'll present solutions to this class of problems using TOP and APPLY. You will find that these solutions are dramatically simpler than the ones I presented previously, and in some cases they are substantially faster. Indexing guidelines, though, remain the same. That is, you want an index with the key list being the partitioning columns (*EmployeeID*), sort columns (*OrderDate*), tiebreaker columns (*OrderID*), and for covering purposes, the other columns mentioned in the query as the included column list (*CustomerID* and *RequiredDate*). Of course, in SQL Server 2000, which doesn't support included nonkey columns (INCLUDE clause), you need to add those to the key list.

Before going over the different solutions, run the following code to create the desired indexes on the Orders and [Order Details] tables that participate in my examples:

```
CREATE UNIQUE INDEX idx_eid_od_oid_i_cid_rd
    ON dbo.Orders(EmployeeID,  OrderDate, OrderID)
     INCLUDE (CustomerID,  RequiredDate);

CREATE UNIQUE INDEX idx_oid_qtyd_pid
    ON dbo.[Order Details](OrderID,  Quantity DESC,  ProductID);
```

The first solution that I'll present will find the most recent order for each employee. The solution queries the Orders table, filtering only orders that have an *OrderID* value that is equal to the result of a subquery. The subquery returns the *OrderID* value of the most recent order for the current employee by using a simple TOP(1) logic. Listing 7-1 has the solution query, generating the output shown in Table 7-8 and the execution plan shown in Figure 7-1.

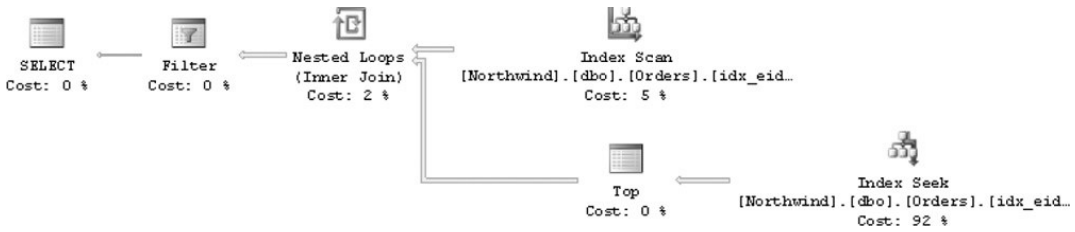

Figure 7-1: Execution plan for the query in Listing 7-1

Listing 7-1: Solution 1 to the Most Recent Order for Each Employee problem

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderID =
      (SELECT TOP(1)OrderID
      FROM dbo.Orders AS O2
      WHERE O2.EmployeeID = O1.EmployeeID
      ORDERBY OrderDate DESC, OrderID DESC);
```

Table 7-8: Most Recent Order for Each Employee

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate
11077	RATTC	1	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000
11073	PERIC	2	1998-05-05 00:00:00.000	1998-06-02 00:00:00.000
11063	HUNGO	3	1998-04-30 00:00:00.000	1998-05-28 00:00:00.000
11076	BONAP	4	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000
11043	SPECD	5	1998-04-22 00:00:00.000	1998-05-20 00:00:00.000
11045	BOTTM	6	1998-04-23 00:00:00.000	1998-05-21 00:00:00.000

11074	SIMOB	7	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000
11075	RICSU	8	1998-05-06 00:00:00.000	1998-06-03 00:00:00.000
11058	BLAUS	9	1998-04-29 00:00:00.000	1998-05-27 00:00:00.000

This solution has several advantages over the solutions I presented earlier in the book. Compared to the ANSI subqueries solution I presented in Chapter 4, this one is faster and much simpler, especially when you have multiple sort/tiebreaker columns; you simply extend the ORDER BY list in the subquery to include the additional columns. Compared to the solution based on aggregations I presented in Chapter 6, this solution is slower but substantially simpler.

Examine the query's execution plan, which is shown in Figure 7-1. The Index Scan operator shows that the covering index idx_eid_od_oid_i_cid_rd is scanned once. The bottom branch of the Nested Loops operator represents the work done for each row of the Index Scan. Here you see that for each row of the Index Scan, an Index Seek and a Top operation take place to find the given employee's most recent order. Remember that the index leaf level holds the data sorted by *EmployeeID, OrderDate, OrderID*, in that order; this means that the last row within each group of rows per employee represents the sought row. The Index Seek operation reaches the end of the group of rows for the current employee, and the Top operator goes one step backwards to return the key of most recent order. A filter operator then keeps only orders where the outer *OrderID* value matches the one returned by the subquery.

The I/O cost of this query is 1,787 logical reads, and this number breaks down as follows: the full scan of the covering index requires 7 logical reads, because the index spans 7 data pages; each of the 830 index seeks requires at least 2 logical reads, because the index has 2 levels, and some of the index seeks require 3 logical reads in all, since the seek might lead to the beginning of one data page and the most recent *OrderID* might be at the end of the preceding page.

Realizing that a separate seek operation within the index was invoked for each outer order, you can figure out that there's room for optimization here. The performance potential is to invoke only a single seek per employee, not order, because ultimately you are after the most recent order for each employee. I'll describe how to achieve such optimization shortly. But before that, I'd like to point out another advantage of this solution over the ones I presented earlier in the book. Previous solutions were limited to returning only a single order per employee. This solution, however, can be easily extended to support any number of orders per employee, by converting the equality operator to an IN predicate. The solution query is shown in Listing 7-2.

Listing 7-2: Solution 1 to the Most Recent Orders for Each Employee problem

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderIDIN
      (SELECT TOP(3)OrderID
      FROM dbo.Orders AS O2
      WHERE O2.EmployeeID  = O1.EmployeeID
      ORDER BY OrderDate DESC, OrderIDDESC);
```

Now, let's go to the optimization technique. Remember you are attempting to give the optimizer a hint that you want one index seek operation per employee, not one per order. You can achieve this by querying the Employees table and retrieving the most recent *OrderID* for each employee. Create a derived table out of this query against Employees, and join the derived table to the Orders table on matching *OrderID* values. Listing 7-3 has the solution query, generating the execution plan shown in Figure 7-2.

Listing 7-3: Solution 2 to the Most Recent Order for Each Employee problem

```
SELECT O.OrderID, CustomerID, O.EmployeeID, OrderDate, RequiredDate
FROM(SELECT EmployeeID,
           (SELECT TOP(1)OrderID
           FROM dbo.Orders AS O2
           WHERE O2.EmployeeID = E.EmployeeID
           ORDER BY OrderDate DESC, OrderID DESC) AS TopOrder
       FROM dbo.Employees AS E) AS EO
    JOIN dbo.Orders AS 0
        ON O.OrderID= EO.TopOrder;
```

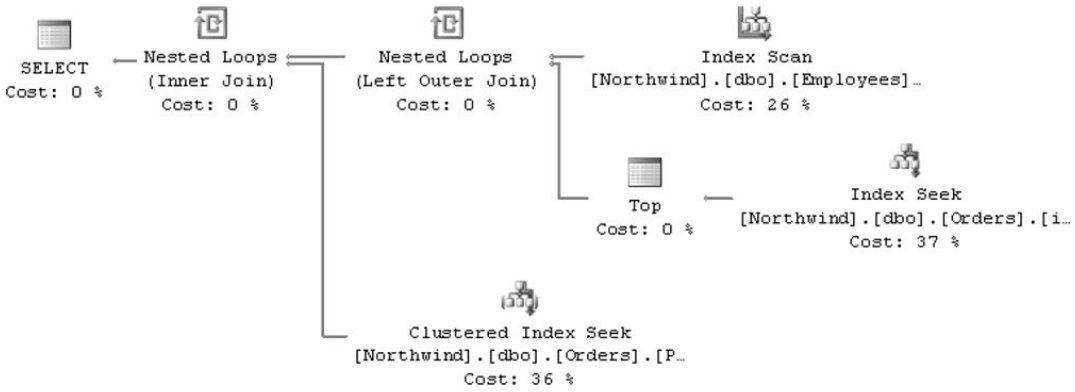

Figure 7-2: Execution plan for the query in Listing 7-3

You can see in the plan that one of the indexes on the Employees table is scanned to access the *EmployeeIDs.* The next operator that appears in the plan (Nested Loops) drives a seek in the index on Orders to retrieve the ID of the employee's most recent order. With 9 employees, only 9 seek operations will be performed, compared to the previous 830 that were driven by the number of orders. Finally, another Nested Loops operator drives one seek per employee in the clustered index on *Orders. OrderID* to look up the attributes of the order from the *OrderID*. If the index on *OrderID* wasn't clustered, you would have seen an additional lookup to access the full data row. The I/O cost of this query is only 36 logical reads.

An attempt to regenerate the same success when you're after more than one order per employee is disappointing. Because you are not able to return more than one key in the SELECT list using a subquery, you might attempt to do something similar in a join condition between Employees and Orders. The solution query is shown in Listing 7-4.

Listing 7-4: Solution 3 to the Most Recent Orders for Each Employee problem

```
SELECT OrderID, CustomerID, E.EmployeeID, OrderDate, RequiredDate
FROM dbo.Employees AS E
     JOIN dbo.Orders AS O1
     ON OrderID IN
       (SELECT TOP(3) OrderID
        FROM dbo.Orders AS   O2
        WHERE O2.EmployeeID = E.EmployeeID
        ORDER BY OrderDate DESC, OrderID DESC);
```

However, this solution yields the poor plan shown in Figure 7-3, generating 15,897 logical reads against the Orders table and 1661 logical reads against the Employees table. In this case, you'd be better off using the solution I showed earlier that supports returning multiple orders per employee.

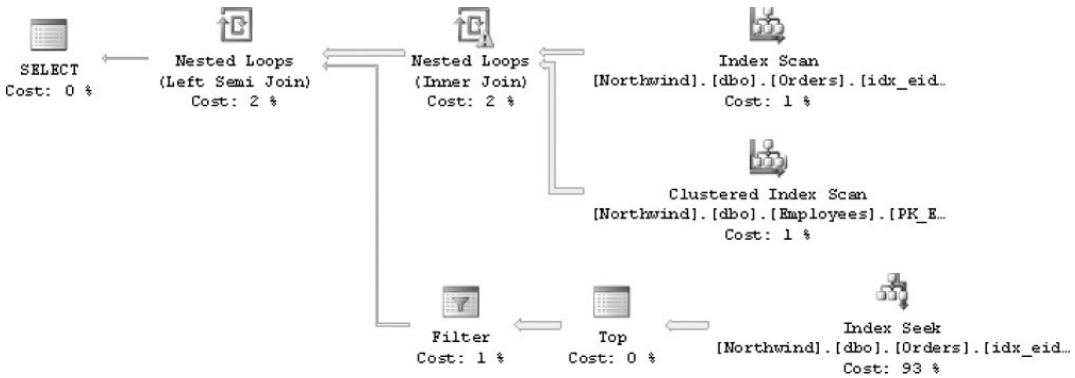

Figure 7-3: Execution plan for the query in Listing 7-4

So far, all solutions were SQL Server 2000 compatible (aside from the fact that in SQL Server 2000 you don't use parentheses with TOP, of course). In SQL Server 2005, you can use the APPLY operator in a solution that outperforms all other solutions I've shown thus far, and that also supports returning multiple orders per employee. You apply to the Employees table a table expression that returns, for a given row of the Employees table, the *n* most recent orders for the employee in that row. Listing 7-5 has the solution query, generating the execution plan shown in Figure 7-4.

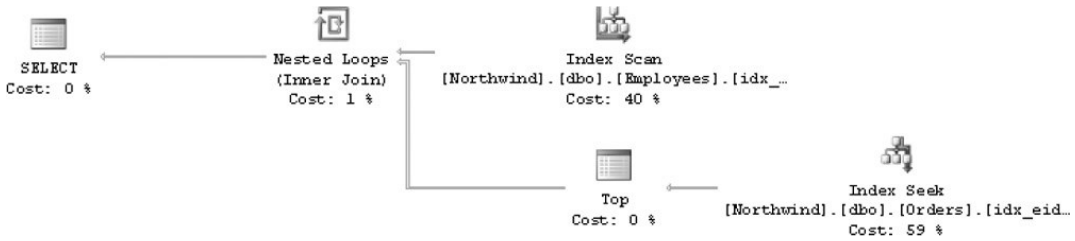

Figure 7-4: Execution plan for the query in Listing 7-5

Listing 7-5: Solution 4 to the Most Recent Orders for Each Employee problem

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Employees AS E
    CROSS APPLY
      (SELECT TOP(3) OrderID, CustomerID, OrderDate, RequiredDate
       FROM dbo.Orders AS O
       WHERE O.EmployeeID = E.EmployeeID
       ORDER BY OrderDate DESC, OrderIDDESC) AS A;
```

The plan scans an index on the Employees table for the *EmployeeID* values. Each *EmployeeID* value drives a single seek within the covering index on Orders to return the requested most recent 3 orders for that employee. The interesting part here is that you don't get only the keys of the rows found; rather, this plan allows for returning multiple attributes. So there's no need for any additional activities to return the non-key attributes. The I/O cost of this query is only 18 logical reads.

Surprisingly, there's a solution that can be even faster than the one using the APPLY operator in certain circumstances that I'll describe shortly. The solution uses the ROW_NUMBER function. You calculate the row number of each order, partitioned by *EmployeeID*, and based on *OrderDate* DESC, *OrderID* DESC order. Then, in an outer query, you filter only results with a row number less than or equal to 3. The optimal index for this solution is similar to the covering index created earlier, but with the *OrderDate* and *OrderID* columns defined in descending order:

```
CREATE UNIQUE INDEX idx_eid_odD_oidD_i_cid_rd
    ON dbo.Orders(EmployeeID, OrderDate DESC, OrderID DESC)
     INCLUDE(CustomerID, RequiredDate);
```

Listing 7-6 has the solution query, generating the execution plan shown in Figure 7-5.

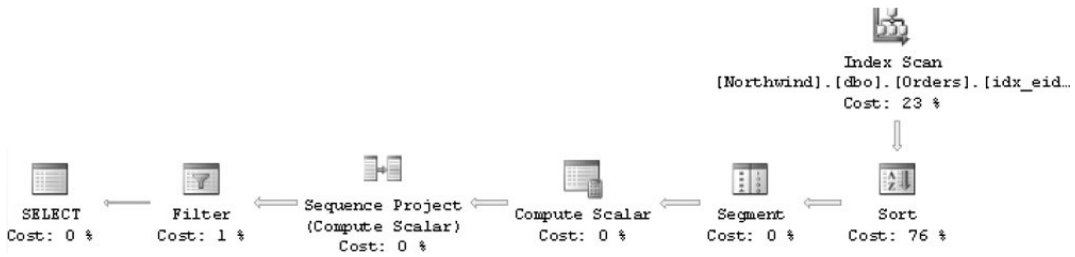

Figure 7-5: Execution plan for the query in Listing 7-6

Listing 7-6: Solution 5 to the Most Recent Orders for Each Employee Problem

```
SELECT OrderID, CustomerID, OrderDate, RequiredDate
FROM (SELECTOrderID, CustomerID, OrderDate, RequiredDate,
        ROW_NUMBER()OVER(PARTITION BY EmployeeID
                         ORDER BY OrderDate DESC, OrderIDDESC) AS RowNum
      FROM dbo.Orders)AS D
WHERE RowNum < = 3;
```

I already described the execution plans generated for ranking functions in Chapter 4, and this one is very similar. The I/O cost here is only 7 logical reads caused by the single full scan of the covering index. Note that to calculate the row numbers here, the index must be fully scanned. With large tables, when you're seeking a small percentage of rows per group, the APPLY operator will be faster because the total cost of the multiple seek operations, one per group, will be

lower than a full scan of the covering index.

There's an important advantage that the solutions using the APPLY operator and the ROW_NUMBER function have over the SQL Server 2000–compatible solutions using TOP. The SQL Server 2000–compatible solutions are supported only when the table at hand has a single column key because they rely on a subquery returning a scalar. The new solutions, on the other hand, are just as applicable with composite keys. For example, say you were after the top 3 order details for each order, with precedence determined by *Quantity* DESC, and where *ProductID* ASC is used as the tiebreaker ordering. The [Order Details] table has a composite primary key, (*OrderID, ProductID*), so you can't return a key for this table from a subquery. On the other hand, the APPLY operator doesn't rely on having a single-column key. It cares only about the correlation of the inner [Order Details] table to the outer Orders table based on *OrderID* match and on a sort based on *Quantity* DESC and *ProductID* ASC:

```
SELECT D.OrderID, ProductID, Quantity
FROM dbo.Orders AS O
   CROSSAPPLY
    (SELECT TOP(3)OD.OrderID, ProductID, Quantity
     FROM  [Order Details] AS OD
     WHERE OD.OrderID = O.OrderID
     ORDER BY Quantity DESC, ProductID) AS D;
```

Similarly, the ROW_NUMBER–based solution doesn't rely on having a single-column key. It simply calculates row numbers partitioned by *OrderID*, sorted by *Quantity* DESC and *ProductID* ASC:

```
SELECT OrderID, ProductID, Quantity
FROM(SELECTROW_NUMBER() OVER(PARTITION BY OrderID
                             ORDER BY Quantity DESC, ProductID) AS RowNum,
       OrderID, ProductID, Quantity
      FROM dbo.[OrderDetails]) AS D
WHERE RowNum < = 3;
```

Matching Current and Previous Occurrences

Matching current and previous occurrences is yet another problem for which you can use the TOP option. The problem is matching to each "current" row, a row from the same table that is considered the "previous" row based on some ordering criteria—typically, time based. Such a request serves the need to make calculations involving measurements from both a "current" row and a "previous" row. Examples for such requests are calculating trends, differences, ratios, and so on. When you need to include only one value from the previous row for your calculation, use a simple TOP(1) subquery to get that value. But when you need multiple measurements from the previous row, it makes more sense in terms of performance to use a join, rather than multiple subqueries.

Suppose you need to match each employee's order with her previous order, using *OrderDate* to determine the previous order and using *OrderID* as a tiebreaker. Once the employee's orders are matched, you can request calculations involving attributes from both sides—for example, calculating differences between the current and previous order dates, required dates, and so on. For brevity's sake, I won't be showing the actual calculations of differences; rather, I'll just focus on the matching techniques. One solution is to join two instances of the Orders table: one representing the current rows (*Cur*), and the other representing the previous row (*Prv*). The join condition will match *Prv.OrderID* with the *OrderID* representing the previous order, which you return from a TOP(1) subquery. You use a LEFT OUTER join to keep the "first" order for each employee. An inner join would eliminate such orders because a match will not be found for them. Listing 7-7 has the solution query to the matching problem.

Listing 7-7: Query Solution 1 to the Matching Current and Previous Occurrences problem

```
SELECT Cur.EmployeeID,
    Cur.OrderID AS CurOrderID, Prv.OrderID AS PrvOrderID,
    Cur.OrderDate AS CurOrderDate, Prv.OrderDate AS PrvOrderDate,
    Cur.RequiredDate AS CurReqDate, Prv.RequiredDate AS PrvReqDate
FROM dbo.Orders AS   Cur
    LEFT OUTERJOIN dbo.Orders AS Prv
        ON Prv.OrderID =
       (SELECT TOP(1) OrderID
        FROM dbo.Orders AS O
        WHERE O.EmployeeID = Cur.EmployeeID
          AND  (O.OrderDate  < Cur.OrderDate
                OR  (O.OrderDate = Cur.OrderDate
                    AND O.OrderID  < Cur.OrderID))
```

```
        ORDER BY OrderDate DESC, OrderID DESC)
ORDER BY Cur.EmployeeID, Cur.OrderDate, Cur.OrderID;
```

The subquery's filter is a bit tricky because ordering/precedence is determined by two attributes: *OrderDate* (ordering column) and *OrderID* (tiebreaker). Had the request been for precedence based on a single column—say, *OrderID* alone—the filter would have been much simpler—*O. OrderID < Cur.OrderID*. Because two attributes are involved, "previous" rows are identified with a logical expression that says: *inner_sort_col < outer_sort_col or (inner_sort_col = outer_sort_col and inner_tiebreaker < outer_tiebreaker)*.

This query generates the execution plan shown in Figure 7-6, with an I/O cost of 3,533 logical reads.

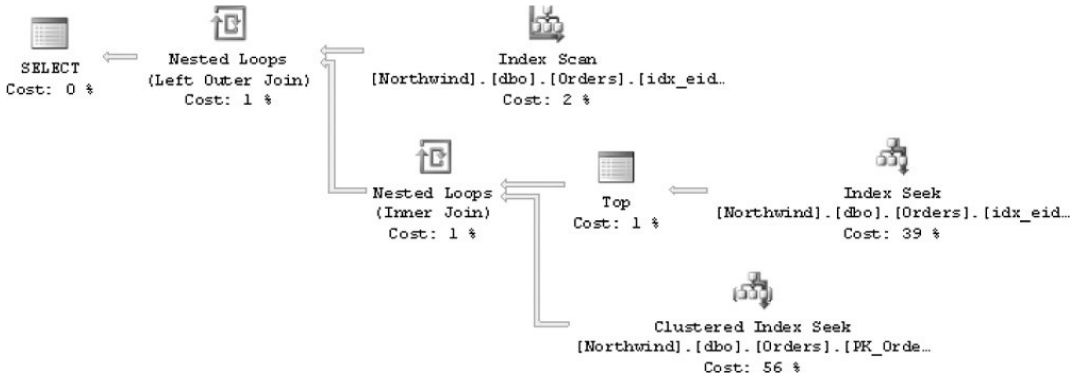

Figure 7-6: Execution plan for the query in Listing 7-7

The plan first scans the covering index I created earlier on the key list (*EmployeeID*, *OrderDate*, *OrderID*), with the covered columns (*CustomerID*, *RequiredDate*) specified as included columns. This scan's purpose is to return the "current" rows. For each current row, a Nested Loops operator initiates an Index Seek operation in the same index, driven by the subquery to fetch the key (*OrderID*) of the "previous" row. For each returned previous *OrderID*, another Nested Loops operator retrieves the requested list of attributes of the previous row. You realize that one of the two seek operations is superfluous and there's potential for a revised query that would issue only one seek per current order.

In SQL Server 2000, you can try various query revisions that might improve performance. But because per outer row, a request with the TOP option can be initiated only in a subquery and not in a table expression, it is not simple to avoid two different activities per current order using TOP. Listing 7-8 has an example of a query revision that allows for the optimized plan shown in Figure 7-7, when the query is run in SQL Server 2005.

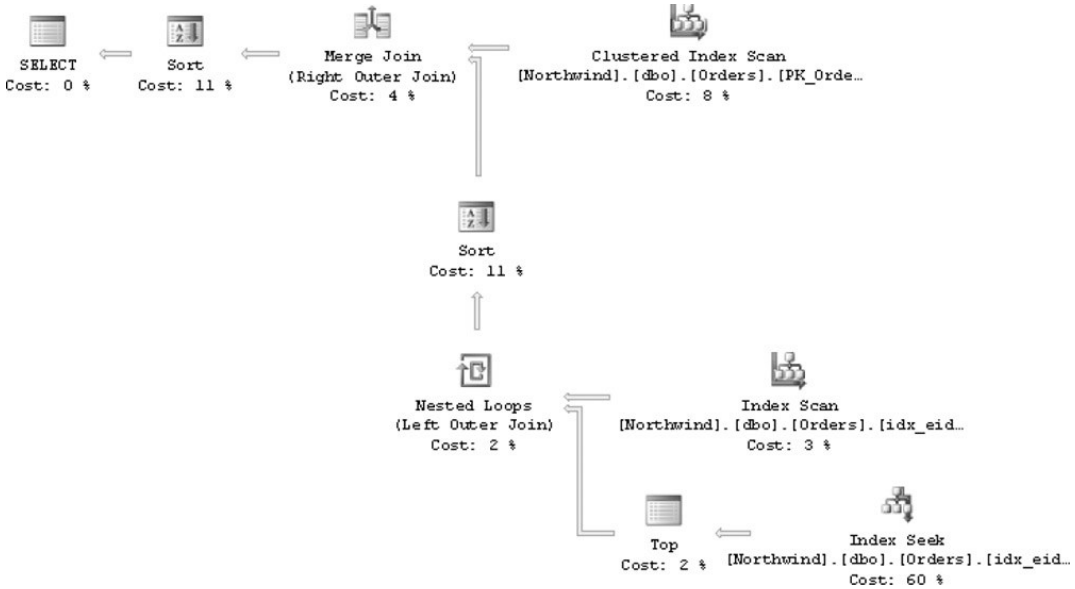

Figure 7-7: Execution plan for the query in Listing 7-8

Listing 7-8: Query Solution 2 to the Matching Current and Previous Occurrences problem

```
SELECT Cur.EmployeeID,
    Cur.OrderID AS CurOrderID, Prv.OrderID AS PrvOrderID,
    Cur.OrderDate AS CurOrderDate, Prv.OrderDate AS PrvOrderDate,
    Cur.RequiredDate AS CurReqDate, Prv.RequiredDate AS PrvReqDate
FROM (SELECTEmployeeID, OrderID, OrderDate, RequiredDate,
        (SELECT TOP(1) OrderID
         FROM dbo.Orders AS O2
         WHERE O2.EmployeeID = O1.EmployeeID
           AND (O2.OrderDate < O1.OrderDate
               OR O2.OrderDate = O1.OrderDate
                  AND O2.OrderID < O1.OrderID)
         ORDER BY OrderDate DESC, OrderID DESC) AS PrvOrderID
      FROM dbo.Orders AS O1)ASCur
    LEFT OUTER JOIN dbo.Orders AS Prv
        ON Cur.PrvOrderID = Prv.OrderID
ORDER BY Cur.EmployeeID, Cur.OrderDate, Cur.OrderID;
```

This plan incurs an I/O cost of 2,033 logical reads (though in SQL Server 2000 you get a different plan with a higher I/O cost). The solution creates a derived table called *Cur* that contains current orders, with an additional column (*PrvOrderID*) holding the *OrderID* of the previous order as obtained by a correlated subquery. The outer query then joins *Cur* with another instance of Orders, aliased as *Prv*, which supplies the full list of attributes from the previous order. The performance improvement and the lower I/O cost is mainly the result of the Merge join algorithm that the plan uses. In the graphical query plan, the upper input to the Merge Join operator is the result of an ordered scan of the clustered index on *OrderID*, representing the "previous" orders, and this is the nonpreserved side of the outer join. The lower input is the result of scanning the covering index and fetching each previous *OrderID* with a seek operation followed by a Top 1. To prepare this input for a merge, the plan sorts the rows by *OrderID*.

A merge join turned out to be cost-effective here because the rows of the Orders table were presorted on the clustered index key column *OrderID* and it was not too much work to sort the other input in preparation for the merge. In larger production systems, things will most likely be different. With a much larger number of rows and a different clustered index—on a column that frequently appears in range queries, perhaps—you shouldn't expect to see the same query plan.

> **Note** Keep in mind that the value of the performance discussions that I'm conducting in the book is in understanding how to read plans and how to write queries in more than one way. You might get different execution plans in different versions of SQL Server—as is the case with the query in Listing 7-8. The last paragraph described the plan that you get in SQL Server 2005, which is different than the one you would get in SQL Server 2000. Similarly, execution plans can change between service pack levels of the product. Also, remember that execution plans might vary when data distribution changes.

This is where the APPLY operator comes in handy. It often leads to simple and efficient plans that perform well even with large volumes of data. Using the APPLY operator in this case leads to a plan that scans the data once to get the current orders and performs a single index seek for each current order to fetch from the covering index all the attributes of the previous order at once.

Listing 7-9 has the solution query, which generates the plan shown in Figure 7-8, with an I/O cost of 2,011 logical reads.

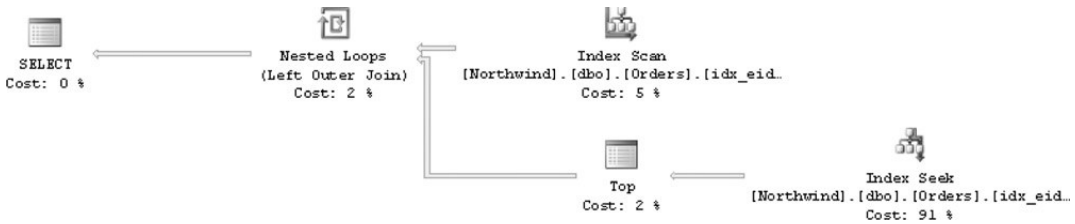

Figure 7-8: Execution plan for the query in Listing 7-9

Listing 7-9: Query Solution 3 to the Matching Current and Previous Occurrences problem

```
SELECT Cur.EmployeeID,
    Cur.OrderID AS CurOrderID, Prv.OrderID AS PrvOrderID,
    Cur.OrderDate AS CurOrderDate, Prv.OrderDate AS PrvOrderDate,
    Cur.RequiredDate AS CurReqDate, Prv.RequiredDate AS PrvReqDate
```

```
FROM dbo.Orders AS Cur
    OUTER APPLY
        (SELECT TOP(1) OrderID, OrderDate, RequiredDate
      FROM dbo.Orders AS O
      WHERE O.EmployeeID = Cur.EmployeeID
        AND (O.OrderDate < Cur.OrderDate
             OR (O.OrderDate = Cur.OrderDate
                AND O.OrderID < Cur.OrderID))
      ORDERBY OrderDate DESC, OrderID DESC) AS Prv
ORDER BY Cur.EmployeeID, Cur.OrderDate, Cur.OrderID;
```

But if you're not satisfied with "reasonable" performance and want a solution that really rocks, you will need the ROW_NUMBER function. You can create a CTE that calculates row numbers for orders partitioned by *EmployeeID* and based on *OrderDate*, *OrderID* ordering. Join two instances of the CTE, one representing the current orders and the other representing the previous orders. The join condition will be based on matching *EmployeeID* values and row numbers that differ by one. Listing 7-10 has the solution query, generating the execution plan shown in Figure 7-9.

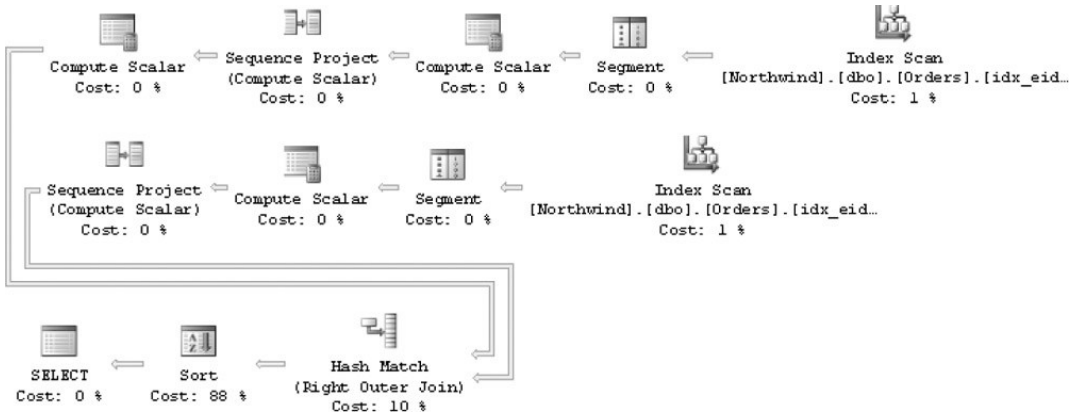

Figure 7-9: Execution plan for the query in Listing 7-10

Listing 7-10: Query Solution 4 to the Matching Current and Previous Occurrences problem

```
WITH OrdersRN AS
(
    SELECT EmployeeID, OrderID, OrderDate, RequiredDate,
        ROW_NUMBER()OVER(PARTITION BY EmployeeID
                    ORDER BY OrderDate, OrderID) AS rn
    FROM dbo.Orders
)
SELECT Cur.EmployeeID,
    Cur.OrderID AS CurOrderID, Prv.OrderID AS PrvOrderID,
    Cur.OrderDate AS CurOrderDate, Prv.OrderDate AS PrvOrderDate,
    Cur.RequiredDate AS CurReqDate, Prv.RequiredDate AS PrvReqDate
FROM OrdersRN AS Cur
    LEFT OUTER JOIN OrdersRN AS Prv
        ONCur.EmployeeID = Prv.EmployeeID
        AND Cur.rn = Prv.rn+  1
ORDER BY Cur.EmployeeID, Cur.OrderDate, Cur.OrderID;
```

Because the plan only scans the covering index twice to access the order attributes and calculate the row numbers, it incurs a total I/O cost of 14 logical reads, leaving all other solutions lagging far behind in terms of performance.

To clean up, run the following code, which drops indexes used for the solutions presented here:

```
DROP INDEX dbo.Orders.idx_eid_od_oid_i_cid_rd;
DROP INDEX dbo.Orders.idx_eid_odD_oidD_i_cid_rd;
DROP INDEX dbo.[Order Details].idx_oid_qtyd_pid;
```

Paging

I started talking about paging in Chapter 4, where I presented solutions based on row numbers. As a reminder, you're looking to return rows from the result set of a query in pages or chunks, allowing the user to navigate through the pages. In my examples, I used the Orders table in the Northwind database.

In production environments, paging typically involves dynamic filters and sorting based on user requests. To focus on the paging techniques, I'll assume no filters here and a desired order of *OrderDate* with *OrderID* as a tiebreaker.

The optimal index for the paging solutions that I'll present follows similar guidelines to other TOP solutions I presented—that is, an index on the sort column or columns and the tiebreaker column or columns. If you can afford to, make the index a covering index, either by making it the table's clustered index, or if it is nonclustered, by including the other columns mentioned in the query. Remember from Chapter 3 that in SQL Server 2005 an index can contain nonkey columns, which are specified in the INCLUDE clause of the CREATE INDEX command. The non-key columns of an index appear only in the leaf level of the index. In SQL Server 2000, nonclustered indexes cannot include non-key columns. You must add additional columns to the key list if you want the index to cover the query, or you must make the index the table's clustered index. If you cannot afford a covering index, at least make sure that you create one on the *sort+tiebreaker* columns. The plans will be less efficient than with a covering one because lookups will be involved to obtain the data row, but at least you won't get a table scan for each page request.

For sorting by *OrderDate* and *OrderID*, and to cover the columns *CustomerID* and *EmployeeID*, create the following index:

```
CREATE INDEX idx_od_oid_i_cid_eid
   ON dbo.Orders(OrderDate, OrderID) INCLUDE(CustomerID, EmployeeID);
```

Note In SQL Server 2000, remember to make all columns part of the key list, as the INCLUDE clause was introduced in SQL Server 2005.

The solution I'll present here supports paging through consecutive pages. That is, you request the first page and then proceed to the next. You might also want to provide the option to request a previous page. It is strongly recommended to implement the first, next, and previous page requests as stored procedures for both performance and encapsulation reasons. This way you can get efficient plan reuse, and you can always alter the implementation of the stored procedures if you find more efficient techniques, without affecting the users of the stored procedures.

First Page

Implementing the stored procedure that returns the first page is really simple because you don't need an anchor to mark the starting point. You simply return the number of rows requested from the top, like so:

```
CREATE PROCdbo.usp_firstpage
    @n AS INT = 10
AS
SELECT TOP(@n) OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY OrderDate, OrderID;
GO
```

Note In this example, ORDER BY has two purposes: to specify what TOP means, and to control the order of rows in the result set.

Having an index on the sort columns, especially if it's a covering one like I created for this purpose, allows for an optimal plan where only the relevant page of rows is scanned within the index in order. You can see this by running the following stored procedure and examining the plan shown in Figure 7-10:

```
EXEC dbo.usp_firstpage;
```

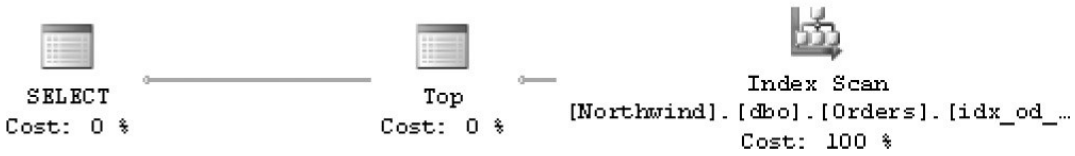

Figure 7-10: Execution plan for stored procedure dbo.usp_firstpage

Rows are scanned within the index, starting with the head of the linked list and moving forward in an ordered fashion. The

Top operator stops the scan as soon as the requested number of rows was accessed.

Next Page

The request for a "next" page has to rely on some anchor row that marks where the page should start. This anchor should be provided to the stored procedure as input. The anchor could be the sort column values of the last row on the previous page because, as you might remember, for determinism purposes the sort values must be unique. In the client application, you already retrieved the previous page. So you can simply set aside the sort column values from the last row in the previous page. When you get a request for the next page, you can provide those as an input to the stored procedure.

Bearing in mind that, in practice, filters and sorting are usually dynamic, you can't rely on any particular number or type of columns as input parameters. So a smarter design, which would accommodate later enhancement of the procedure to support dynamic execution, would be to provide the primary key as input, and not the sort column values. The client application would set aside the primary key value from the last row it retrieved and use it as input to the next invocation of the stored procedure.

Here's the implementation of the usp_nextpage stored procedure:

```
CREATE PROC dbo.usp_nextpage
    @anchor AS INT, -- key of last row in prev page
    @n AS INT = 10
AS
SELECT TOP(@n) O.OrderID, O.OrderDate, O.CustomerID, O.EmployeeID
FROM dbo.Orders AS O
    JOIN dbo.Orders AS A
        ON A.OrderID = @anchor
        AND (O.OrderDate > A.OrderDate
         OR (O.OrderDate = A.OrderDate
             AND O.OrderID > A.OrderID))
ORDER BY O.OrderDate, O.OrderID;
GO
```

The procedure joins the two instances of the orders table: one called O, representing the next page, and one called A, representing the anchor. The join condition first filters the anchor instance with the input key, and then it filters the instance representing the next page so that only rows following the anchor will be returned. The columns *OrderDate* and *OrderID* determine precedence both in terms of the logical expression in the ON clause that filters rows following the anchor, and in terms of the ORDER BY clause that TOP relies on. To test the stored procedure, first execute it with the *OrderID* from the last row returned from the first page (10257) as the anchor. Then execute it again with the *OrderID* of the last row in the second page (10267) as the anchor:

```
EXEC dbo.usp_nextpage @anchor = 10257;
EXEC dbo.usp_nextpage @anchor = 10267;
```

Remember that the client application iterates through the rows it got back from SQL Server, so naturally it can pick up the key from the last row and use it as input to the next invocation of the stored procedure.

Both procedure calls yield the same execution plan, which is shown in Figure 7-11.

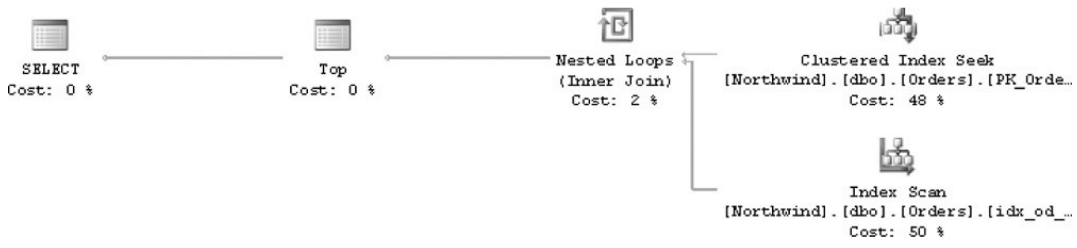

Figure 7-11: Execution plan for the stored procedure usp_nextpage

You will see a single seek operation within the clustered index to fetch the anchor row, followed by an ordered scan within the covering index to fetch the next page of rows. That's not a very efficient plan. Ideally, the optimizer would have performed a seek within the covering index to the first row from the desired page of orders; then it would have followed with a partial ordered scan to grab the rest of the rows in the desired page of orders, physically accessing only the relevant rows. The reason for getting an inefficient plan is because the filter has an OR operator between the expression *O. OrderDate > A. OrderDate*, and the expression *O. OrderDate = A. OrderDate AND O. OrderID > A.OrderID*. See the

sidebar "Logical Transformations" for details about OR optimization vs. AND optimization. Following the sidebar, I'll provide the optimized implementation of the stored procedure using AND logic.

Logical Transformations

In several solutions I've presented, I used logical expressions with an OR operator to deal with precedence based on multiple attributes. Such was the case in the recent solutions for paging, matching current and previous occurrences, and other problems. I used OR logic because this is how human minds are accustomed to thinking. The logical expressions using OR logic are fairly intuitive for the purpose of determining precedence and identifying rows that follow a certain anchor.

However, because of the way SQL Server's optimizer works, OR logic is problematic in terms of performance, especially when some of the filtered columns are not indexed. Consider for example a filter such as *col1 = 5 OR col2 = 10*. If you have individual indexes on *col1* and *col2*, the optimizer can filter the rows in each index and then perform an index intersection between the two. However, if you have an index on only one of the columns, even when the filter is very selective, the index is useless. SQL Server would still need to scan the whole table to see whether rows that didn't match the first filter qualify for the second condition.

On the other hand, AND logic has much better performance potential. With each expression, you narrow down the result set. Rows filtered by one index are already a superset of the rows you'll end up returning. So an index on any of the filtered columns can potentially be used to advantage. Whether or not it is worthwhile to use the existing index is a matter of selectivity, but the potential is there. For example, consider the filter *col1 = 5 AND col2 = 10*. The optimal index here is a composite one created on both columns. However, if you have an index on only one of them, and it's selective enough, that's sufficient already. SQL Server can filter the data through that index, and then look up the rows and examine whether they also meet the second condition.

In this chapter, the logical expressions I used in my solutions used OR logic to identify rows following a given anchor. For example, say you're looking at the row with an *OrderID* of 11075, and you're supposed to identify the rows that follow, where precedence is based on *OrderDate* and *OrderID* is the tiebreaker. The *OrderDate* of the anchor is '19980506'. A query returning the rows that come after this anchor row is very selective. I used the following logic to filter these rows:

```
OrderDate > '19980506' OR (OrderDate = '19980506' AND OrderID > 11075)
```

Say that you could afford creating only one index, on *OrderDate.* Such an index is not sufficient in the eyes of the optimizer to filter the relevant rows because the logical expression referring to *OrderDate* is followed by an OR operator, with the right side of the operator referring to other columns (*OrderID*, in this case). Such a filter would yield a table scan. You can perform a logical transformation here and end up with an equivalent expression that uses AND logic. Here's the transformed expression:

```
OrderDate > = '19980506' AND (OrderDate > '19980506' OR OrderID > 11075)
```

Instead of specifying *OrderDate > '19980506'*, you specify *OrderDate > = '19980506'*. Now you can use an AND operator and request either rows where the *OrderDate* is greater than the anchor's *OrderDate* (meaning the *OrderDate* is not equal to the anchor's *OrderDate*, in which case you don't care about the value of *OrderID*); or the *OrderID* is greater than the anchor's *OrderID* (meaning the *OrderDate* is equal to the anchor's *OrderDate*). The logical expressions are equivalent. However, the transformed one has the form *OrderDate_comparison* AND *other_logical_expression*—meaning that now an index on *OrderDate* alone can be considered. To put these words into action, first create a table called MyOrders containing the same data as the Orders table, and an index only on *OrderDate*:

```
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
    DROP TABLE dbo.MyOrders;
GO
SELECT *INTO dbo.MyOrders FROM dbo.Orders
CREATE INDEX idx_dt ON dbo.MyOrders(OrderDate);
```

Next, run the query in Listing 7-11, which uses OR logic, and examine the plan shown in Figure 7-12.

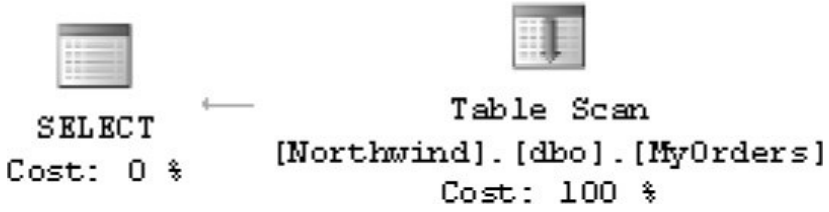

Figure 7-12: Execution plan for the query in Listing 7-11

Listing 7-11: Query using OR logic

```
SELECT OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.MyOrders
WHERE OrderDate > '19980506'
      OR (OrderDate = '19980506'AND OrderID > 11075);
```

You will see a table scan, which in the case of this table costs 21 logical reads. Of course, with more realistic table sizes you will see substantially more I/O.

Next, run the query in Listing 7-12, which uses AND logic, and examine the plan shown in Figure 7-13.

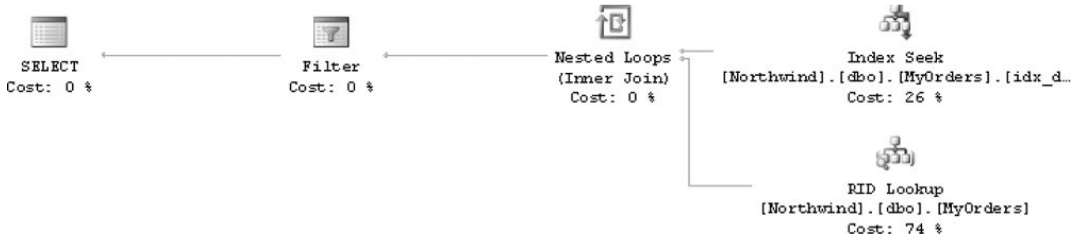

Figure 7-13: Execution plan for the query in Listing 7-12

Listing 7-12: Query using AND logic

```
SELECT OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.MyOrders
WHERE OrderDate > = '19980506'
    AND (OrderDate > '19980506'OR OrderID > 11075);
```

You will see that the index on *OrderDate* is used, and the I/O cost of this query is 6 logical reads. Creating an index on both columns (*OrderDate, OrderID*) is even better:

```
CREATE INDEX idx_dt_oid ON dbo.MyOrders(OrderDate, OrderID);
```

Run the query in Listing 7-11, which uses the OR logic. You will see in the plan, shown in Figure 7-14, that the new index is used. The I/O cost for this plan is 6 logical reads.

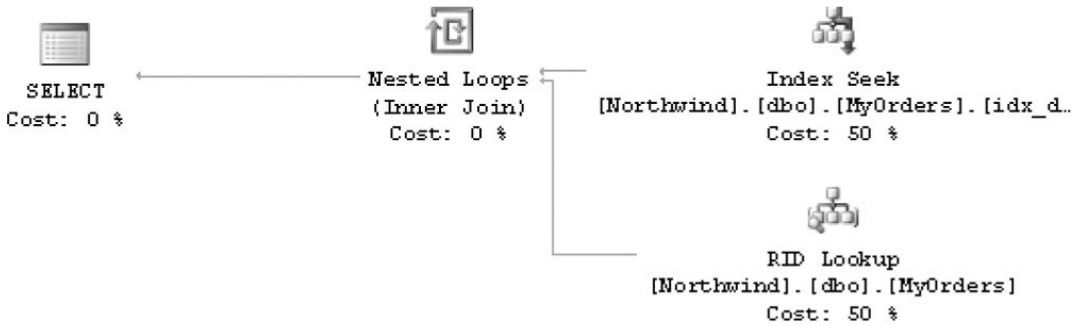

Figure 7-14: Execution plan for the query in Listing 7-11, with the new index in place

Run the query in Listing 7-12, which uses the AND logic. You will see the plan shown in Figure 7-15, which might seem similar, but it yields even a lower I/O cost of only 4 logical reads.

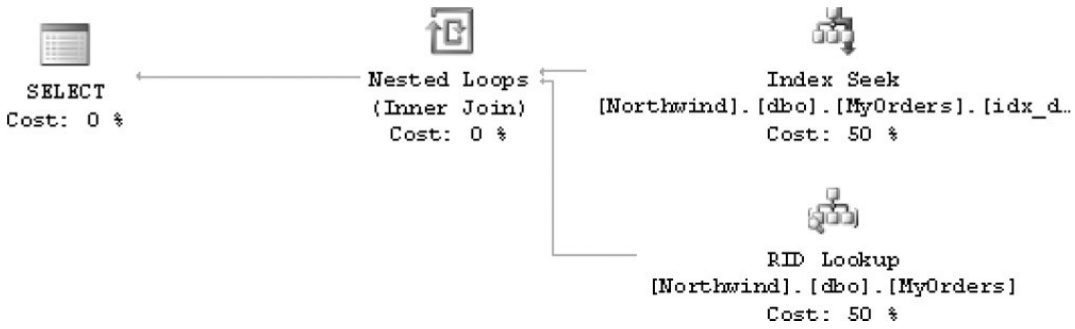

Figure 7-15: Execution plan for the query in Listing 7-12, with the new index in place

The conclusion is, of course, that SQL server can optimize AND logic better than OR logic. All the solutions I presented in this chapter would be better off in terms of performance if you transformed their OR logic to AND logic. Similarly, you might be able to achieve such transformations with other logical expressions.

Another conclusion is that it's better to have an index on all columns determining precedence. The problem is that in production environments you can't always afford it.

Note When discussing subjects that involve logic, I like to use small tables such as those in Northwind, with simple and recognizable data. With such tables, the differences in logical reads that you see when testing your solutions are small. In real performance tests and benchmarks, you should use more realistic table sizes as your test data, such as the test data I used in Chapter 3. For example, using the usp_nextpage procedure, which returns the next page of orders, you will see very small I/O differences between OR logic and the AND logic, as I'll present shortly. But when I tested the solution against an Orders table with about a million rows, the OR implementation costs more than a thousand logical reads, while the AND implementation costs only 11 logical reads, physically accessing only the relevant page of orders.

When you're done, don't forget to get rid of the MyOrders table created for these examples:

```
IF OBJECT_ID('dbo.MyOrders') IS NOT NULL
    DROP TABLE dbo.MyOrders;
```

Back to our usp_nextpage procedure, here's the optimized implementation that transforms the OR logic to AND logic:

```
ALTER PROC dbo.usp_nextpage
    @anchor AS INT, --  key of last row in prev page
    @n AS INT= 10
AS
SELECT TOP(@n) O.OrderID, O.OrderDate, O.CustomerID, O.EmployeeID
FROM dbo.Orders AS O
    JOIN dbo.Orders AS A
        ON A.OrderID = @anchor
        AND(O.OrderDate > = A.OrderDate
         AND(O.OrderDate > A.OrderDate
              OR O.OrderID > A.OrderID))
ORDER BY O.OrderDate, O.OrderID;
GO
```

Notice that the AND expression within the parentheses is logically equivalent to the previous OR expression; I just implemented the techniques described in the Logical Transformations sidebar. To show that the AND implementation is really optimized better, run the following code and examine the execution plan shown in Figure 7-16:

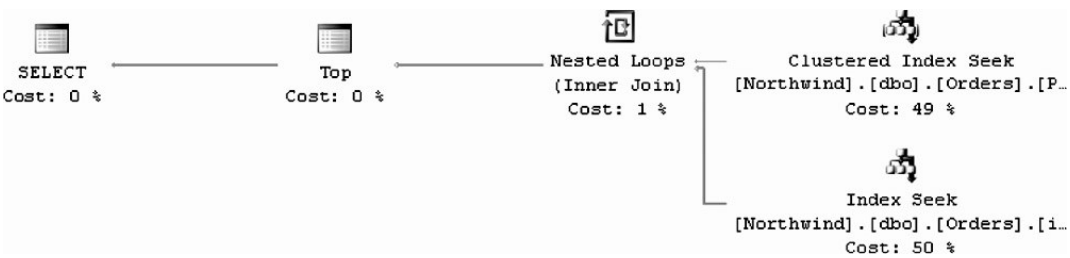

Figure 7-16: Execution plan for the stored procedure usp_nextpage—second version

```
EXEC dbo.usp_nextpage @anchor = 10257;
```

Now you get the desired plan. You see a single seek operation within the clustered index to fetch the anchor row, followed by a seek within the covering index and a partial ordered scan, physically accessing only the relevant rows in the desired page of orders.

Previous Page

There are two approaches to dealing with requests for previous pages. One is to locally cache pages already retrieved to the client. This means that you need to develop a caching mechanism in the client. A simpler approach is to implement another stored procedure that works like the usp_nextpage procedure in reverse. The anchor parameter will be the key of the first row after the page you want. The comparisons within the procedure will use < instead of >, and the TOP clause will use an ORDER BY list that defines the opposite sorting direction.

If these were the only changes, you would get the correct page, but in reverse order from normal. To fix the ordering of the result set, encapsulate the query as a derived table, and apply SELECT … ORDER BY to this derived table, with the desired ordering.

Here's the implementation of the usp_prevpage procedure:

```
CREATE PROC dbo.usp_prevpage
    @anchor AS INT, -- keyof first row in next page
    @n AS INT = 10
AS
SELECT OrderID, OrderDate, CustomerID, EmployeeID
FROM (SELECT TOP(@n) O.OrderID, O.OrderDate, O.CustomerID, O.EmployeeID
      FROM dbo.Orders AS O
        JOIN dbo.Orders AS A
          ON A.OrderID = @anchor
          AND (O.OrderDate < = A.OrderDate
               AND (O.OrderDate < A.OrderDate
                   OR O.OrderID < A.OrderID))
      ORDER BY O.OrderDate DESC, O.OrderIDD ESC) AS D
ORDER BY OrderDate, OrderID;
GO
```

To test the procedure, run it with *OrderID* values from the first rows on the pages you already got:

```
EXEC dbo.usp_prevpage @anchor = 10268;
EXEC dbo.usp_prevpage @anchor = 10258;
```

Examine the execution plan shown in Figure 7-17, produced for the execution of the usp_prevpage procedure.

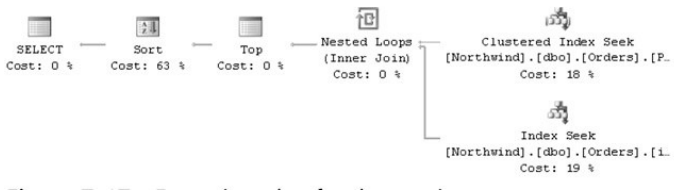

Figure 7-17: Execution plan for the previous page

You will find an almost identical plan to the one produced for the usp_nextpage procedure, with an additional Sort operator, which is a result of the extra ORDER BY clause in the usp_prevpage procedure.

Here I wanted to focus on paging techniques using the TOP option. Remember that the topic is also covered in Chapter 4, where I show paging solutions based on row numbers.

When you're finished, drop the covered index created for the paging solutions:

```
DROP INDEX dbo.Orders.idx_od_oid_i_cid_eid;
```

Tip When using solutions such as the ones in this section, changes in the underlying data will be reflected in requests for new pages. That is, if rows are added, deleted, or updated, new page requests will be submitted against the current version of the data—not to mention the possible failure of the procedure altogether, if the anchor key is

gone. If this behavior is undesirable to you and you'd rather iterate through pages against a static view of the data, you can create a database snapshot right before answering page requests. Submit your paging queries against the snapshot, and get rid of the snapshot as soon as the user finishes.

For details about database snapshots please refer to *Inside Microsoft SQL Server 2005: The Storage Engine* by Kalen Delaney (Microsoft Press, 2006).

Random Rows

This section covers another class of problems that you can solve with the TOP option—returning rows in a random fashion. Dealing with randomness in T-SQL is quite tricky. Typical requests for randomness involve returning a random row from a table, sorting rows in random order, and the like. The first attempt you might make when asked to return a random row might be to use the RAND function as follows:

```
SELECT TOP(1) OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BYRAND();
```

However, if you try running this query several times, you will probably be disappointed to find that you're not really getting a random row. RAND, as well as most other nondeterministic functions (for example, GETDATE) are invoked once per query, not once per row. So you end up getting the same value of RAND for every row, and the ORDER BY clause does not affect the ordering of the query's result set.

Tip You might be surprised to find that the RAND function—when given an integer seed as input—is not really nondeterministic; rather, it's sort of a hash function. Given the same seed, RAND(<seed>) will always yield the same result. For example, run the following code multiple times:

```
SELECT RAND (5);
```

You will always get back 0.713666525097956. And if that's not enough, when you don't specify a seed, SQL Server doesn't really choose a random seed. Rather, the new seed is based on the previous invocation of RAND. Hence, running the following code multiple times will always yield the same two results (0.713666525097956 and 0.454560299686459):

```
SELECT RAND(5);
SELECT RAND();
```

The most important use of RAND(<seed>) is probably to create reproducible sample data, because you can seed it once and then call it repeatedly without a seed to get a well-distributed sequence of values.

If you're seeking a random value, you will have much better success with the following expression:

```
SELECT CHECKSUM(NEWID());
```

Note The NEWID function appears to have good distribution properties; however, to date, I haven't found any documentation from Microsoft that specifies that this is guaranteed or supported.

An interesting behavior of the NEWID function is that unlike other nondeterministic functions, NEWID is evaluated separately for each row if you invoke it in a query. Bearing this in mind, you can get a random row by using the preceding expression in the ORDER BY clause as follows:

```
SELECT TOP(1) OrderID, OrderDate, CustomerID, EmployeeID
FROM dbo.Orders
ORDER BY CHECKSUM(NEWID());
```

This gives me an opportunity to present another example for using the new functionality of TOP, which allows you to specify a self-contained expression as an input. The following query also returns a random row:

```
SELECT TOP(1)OrderID, OrderDate, CustomerID, EmployeeID
FROM (SELECT TOP(100e0*(CHECKSUM(NEWID()) +2147483649)/4294967296e0) PERCENT
        OrderID, OrderDate, CustomerID, EmployeeID
      FROM dbo.Orders
      ORDER BY OrderID) AS D
ORDER BY OrderID DESC;
```

CHECKSUM returns an integer between 2147483648 and 2147483647. Adding 2147483649 and then dividing by the float value 4294967296e0 yields a random number in the range 0 through 1 (excluding 0). Multiplying this random number by 100 returns a random float value greater than 0 and less than or equal to 100. Remember that the TOP PERCENT option

accepts a float percentage in the range 0 through 100, and it rounds up the number of returned rows. A percentage greater than 0 guarantees that at least one row will be returned. The query creating the derived table D thus returns a random number of rows from the table based on *OrderID* (primary key) sort. The outer query then simply returns the last row from the derived table—that is, the one with the greatest *OrderID* values. This solution is not necessarily more efficient than the previous one I presented, but it was a good opportunity to show how the new features of TOP can be used.

With the new APPLY operator, you can now answer other randomness requests easily and efficiently, without the need to explicitly apply iterative logic. For example, the following query returns three random orders for each employee:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Employees AS E

CROSS APPLY
    (SELECT TOP(3) OrderID, CustomerID, OrderDate, RequiredDate
      FROM dbo.Orders AS O
      WHERE O.EmployeeID = E.EmployeeID
      ORDER BY CHECKSUM(NEWID())) AS A;
```

Median

In Chapter 6 in the "Custom Aggregations" section, I discussed techniques to calculate the median value for each group based on the ROW_NUMBER function. Here I'll present techniques relying on TOP that on one hand are slower, but on the other hand are applicable in SQL Server 2000 as well. First run the code in Listing 7-13 to create the Groups table that I used in my previous solutions to obtain a median.

Listing 7-13: Creating and populating the Groups table

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

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

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

Remember that median is the middle value (assuming a sorted list) when the group has an odd number of elements, and it's the average of the two middle values when it has an even number.

It's always a good idea to handle each case separately, and then try to figure out whether the solutions can be merged. So

first, assume there's an odd number of elements. You can use a TOP(50) PERCENT query to access the first half of the elements, including the middle one. Remember that the PERCENT option rounds up. Then simply query the maximum value from the returned result set.

Now handle the even case. The same query you use to get the middle value from an odd number of rows will produce the largest value of the first half of an even number of rows. You can then write a similar query to return the smallest value of the second half. Sum the two values, divide by two, and you have the median in the even case.

Now try to figure out whether the two solutions can be merged. Interestingly, running the solution for the even case against an odd number of elements yields the correct result, because both subqueries used in the even case solution will end up returning the same row when there is an odd number of rows. The average of two values that are equal is obviously the same value.

Here's what the solution looks like when you want to return the median of the *val* column for the whole table:

```
SELECT
    ((SELECT MAX(val)
        FROM (SELECT TOP(50) PERCENT val
           FROM dbo.Groups
           ORDERBY val) AS M1)
      +
      (SELECT MIN(val)
        FROM (SELECT TOP(50) PERCENT val
           FROM dbo.Groups
           ORDER BY val    DESC) AS M2))
    /2.  AS median;
```

To return the median for each group, you need an outer query that groups the data by *groupid*. For each group, you invoke the calculation of the median in a subquery like so:

```
SELECT groupid,
    ((SELECT MAX(val)
        FROM (SELECT TOP(50) PERCENT val
          FROM dbo.Groups AS H1
          WHERE H1.groupid = G.groupid
          ORDER BY val) AS M1)
      +

      (SELECT MIN(val)
        FROM(SELECT TOP(50) PERCENT val
           FROM dbo.Groups AS H2
           WHEREH 2.groupid = G.groupid
           ORDER BY val DESC) AS M2))
    /2.  AS median
FROM dbo.Groups AS G
GROUP BY groupid;
```

This query works in SQL Server 2000 with two small modifications: first, you must write TOP 50 PERCENT instead of TOP (50) PERCENT, and second, to work around an unexpected behavior of queries that use GROUP BY together with subqueries, you must use SELECT DISTINCT instead of GROUP BY to produce just one result row per group. Here is the solution for SQL Server 2000:

```
SELECT DISTINCT groupid,
    ((SELECT MAX(val)
        FROM (SELECT TOP 50 PERCENT val
            FROM dbo.Groups AS H1
            WHERE H1.groupid = G.groupid
            ORDER BY val) AS M1)
      +
      (SELECT MIN(val)
        FROM (SELECT TOP 50 PERCENT val
            FROM dbo.Groups AS H2
            WHEREH 2.groupid = G.groupid
            ORDER BY val DESC) AS M2))
    /2.  AS median
FROM dbo.Groups AS G;
```

Conclusion

As you probably realized from this chapter, TOP and APPLY are two features that in many ways complement each other. TOP with its new capabilities now allows expressions as input and is supported with modification statements. The new TOP functionality replaces the older SET ROWCOUNT option. The new APPLY operator allows for very simple and fast queries, compared to the previous alternatives, whenever you need to apply a table expression to each row of an outer query.