Inside Microsoft SQL Server 2005: T-SQL Querying

by Itzik Ben-Gan, Lubor Kollar and Dejan Sarka

Chapter 4: Subqueries, Table Expressions, and Ranking Functions

Overview

This chapter essentially covers nesting of queries. Within an outer query, you can incorporate inner queries (also known as subqueries). You can use subqueries where a single value is expected (scalar subqueries)—for example, to the right of an equal sign in a logical expression. You can use them where multiple values are expected (multivalued subqueries)—for example, as input to the IN predicate. Or you can use them where a table is expected (table expressions)—for example, in the FROM clause of a query.

I'll refer to scalar and multivalued subqueries just as *subqueries*, and to subqueries that are used where a table is expected as *table expressions*. In this chapter, I'll cover the inline table expressions: derived tables and common table expressions (CTE).

In the last part of the chapter, I'll cover ranking calculations, including row number, rank, dense rank, and tiling. I found it appropriate to cover ranking calculations in this chapter because in Microsoft SQL Server 2000 you can use subqueries for these calculations. SQL Server 2005 introduces built-in ranking functions, which allow you to calculate these values much more simply and efficiently.

Because this book is intended for experienced programmers, I'm assuming that you're already familiar with subqueries and table expressions. I'll go over their definitions briefly, and focus on their applications and on problem solving.

Subqueries

Subqueries can be characterized in two main ways. One is by the expected number of values (either scalar or multivalued), and another is by the subquery's dependency on the outer query (either self-contained or correlated). Both scalar and multivalued subqueries can be either self-contained or correlated.

Self-Contained Subqueries

A self-contained subquery is a subquery that can be run independently of the outer query. Self-contained subqueries are very convenient to debug, of course, compared to correlated subqueries.

Scalar subqueries can appear anywhere in the query where an expression resulting in a scalar value is expected, while multivalued subqueries can appear anywhere in the query where a collection of multiple values is expected.

A scalar subquery is valid when it returns a single value, and also when it returns no values—in which case, the value of the subquery is NULL. However, if a scalar subquery returns more than one value, a run-time error will occur.

For example, run the following code three times: once as shown, a second time with LIKE *N'Kollar'* in place of LIKE *N'Davolio'*, and a third time with LIKE *N'D%*:

```
SET NOCOUNT ON;
USE Northwind;

SELECT OrderID
FROM dbo.Orders
WHERE EmployeeID =
  (SELECT EmployeeID FROM dbo.Employees
  --also try with N'Kollar' and N'D%' in place of N'Davolio'
  WHERE LastName LIKE N'Davolio');
```

With *N'Davolio'*, the subquery returns a single value (1) and the outer query returns all orders with *EmployeeID* 1.

With *N'Kollar'*, the subquery returns no values, and is therefore NULL. The outer query obviously doesn't find any orders for which EmployeeID = NULL and therefore returns an empty set. Note that the query doesn't break (fail), as it's a valid query.

With *N'D%'*, the subquery returns two values (1, 9), and because the outer query expects a scalar, it breaks at run time and generates the following error:

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the subquery follows =,
```

`!=, <, <= ,>, >= or when the subquery is used as an expression.`

Logically, a self-contained subquery can be evaluated just once for the whole outer query. Physically, the optimizer can consider many different ways to achieve the same thing, so you shouldn't think in such strict terms.

Now that we've covered the essentials, let's move on to more sophisticated problems involving self-contained subqueries.

I'll start with a problem belonging to a group of problems called relational division. Relational division problems have many nuances and many practical applications. Logically, it's like dividing one set by another, producing a result set. For example, from the Northwind database, return all customers for whom every Northwind employee from the USA has handled at least one order. In this case, you're dividing the set of all orders by the set of all employees from the USA, and you expect the set of matching customers back. Filtering here is not that simple because for each customer you need to inspect multiple rows to figure out whether you have a match.

Here I'll show a technique using GROUP BY and DISTINCT COUNT to solve relational division problems. Later in the book, I'll show others as well.

If you knew ahead of time the list of all *EmployeeID* s for USA employees, you could write the following query to solve the problem, generating the output shown in Table 4-1:

Table 4-1: Customers with Orders Handled by All Employees from the USA

CustomerID
BERGS
BONAP
ERNSH
FOLKO
HANAR
HILAA
HUNGO
ISLAT
KOENE
LAMAI
LILAS
LINOD
LONEP
MEREP
OLDWO
QUICK
RATTC
SAVEA
TORTU
VAFFE
VICTE
WARTH
WHITC

```
SELECT CustomerID
FROM dbo.Orders
WHERE EmployeeID IN(1, 2, 3, 4, 8)
GROUP BY CustomerID
HAVING COUNT(DISTINCT EmployeeID) = 5;
```

This query finds all orders with one of the five U.S. *EmployeeID* s, groups those orders by *CustomerID*, and returns *CustomerID* s that have (all) five distinct *EmployeeID* values in their group of orders.

To make the solution more dynamic and accommodate lists of *EmployeeID* s that are unknown ahead of time and also large lists even when known, you can use subqueries instead of literals:

```
SELECT CustomerID
FROM dbo.Orders
WHERE EmployeeID IN
  (SELECT EmployeeID FROM dbo.Employees WHERE Country = N'USA')
GROUP BY CustomerID
HAVING COUNT(DISTINCT EmployeeID) =
  (SELECT COUNT(*) FROM dbo.Employees WHERE Country = N'USA');
```

Another problem involving self-contained subqueries is returning all orders placed on the last actual order date of the month. Note that the last actual order date of the month might be different than the last date of the month—for example, if a company doesn't place orders on weekends. So the last actual order date of the month has to be queried from the data. Here's the solution query producing the output (abbreviated) shown in Table 4-2:

Table 4-2: Orders Placed on the Last Actual Order Date of the Month

OrderID	CustomerID	EmployeeID	OrderDate
10461	LILAS	1	1997-02-28
10616	GREAL	1	1997-07-31
10916	RANCH	1	1998-02-27
11077	RATTC	1	1998-05-06
10368	ERNSH	2	1996-11-29
10553	WARTH	2	1997-05-30
10583	WARTH	2	1997-06-30
10686	PICCO	2	1997-09-30
10915	TORTU	2	1998-02-27
10989	QUEDE	2	1998-03-31
…			
11075	RICSU	8	1998-05-06
10687	HUNGO	9	1997-09-30

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders
WHERE OrderDate IN
  (SELECT MAX(OrderDate)
  FROM dbo.Orders
  GROUP BY CONVERT(CHAR(6), OrderDate, 112));
```

The self-contained subquery returns a list of values with the last actual order date of each month, as shown in Table 4-3.

Table 4-3: Last Actual Order Date of Each Month

MAX(OrderDate)
1997-04-30
1996-09-30

1996-10-31
1998-02-27
1997-06-30
1997-08-29
1997-03-31
1997-01-31
1997-09-30
1996-07-31
1997-02-28
1998-01-30
1997-11-28
1998-04-30
1997-05-30
1997-12-31
1998-05-06
1997-10-31
1996-11-29
1996-12-31
1996-08-30
1997-07-31
1998-03-31

The subquery achieves this result by grouping the orders by month and returning the *MAX (OrderDate)* for each group. I extracted the year-plus-month portion of *OrderDate* by converting it to *CHAR(6)* using the style 112 (ISO format 'YYYYMMDD'). Because the target string is shorter than the style's string length (6 characters instead of 8), the two right-most characters are truncated. You could obtain the same result by using *YEAR (OrderDate)*, *MONTH (OrderDate)* as the GROUP BY clause.

The outer query returns all orders with an *OrderDate* that appears in the list returned by the subquery.

Correlated Subqueries

Correlated subqueries are subqueries that have references to columns from the outer query. Logically, the subquery is evaluated once for each row of the outer query. Again, physically, it's a much more dynamic process and will vary from case to case. There isn't just one physical way to process a correlated subquery.

Tiebreaker

I'll start dealing with correlated subqueries through a problem that introduces a very important concept in SQL querying—a *tiebreaker*. I'll refer to this concept throughout the book. A tiebreaker is an attribute or attribute list that allows you to uniquely rank elements. For example, suppose you need the most recent order for each Northwind employee. You are supposed to return only one order for each employee, but the attributes *EmployeeID* and *Order-Date* do not necessarily identify a unique order. You need to introduce a tiebreaker to be able to identify a unique most recent order for each employee. For example, out of the multiple orders with the maximum *OrderDate* for an employee you could decide to return the one with the maximum *OrderID*. In this case, *MAX(OrderID)* is your tiebreaker. Or you could decide to return the row with the maximum *RequiredDate*, and if you still have multiple rows, return the one with the maximum *OrderID*. In this case, your tiebreaker is *MAX(RequiredDate), MAX(OrderID)*. A tiebreaker is not necessarily limited to a single attribute.

Before moving on to the solutions, run the following code to create indexes that support the physical processing of the queries that will follow:

```
CREATE UNIQUE INDEX idx_eid_od_oid
  ON dbo.Orders(EmployeeID, OrderDate, OrderID);
CREATE UNIQUE INDEX idx_eid_od_rd_oid
```

```
  ON dbo.Orders(EmployeeID, OrderDate, RequiredDate, OrderID);
```

I'll explain the indexing guidelines after presenting the solution queries.

Let's start with the basic request to return the orders with the maximum *OrderDate* for each employee. Here you can get multiple rows for each employee because an employee can have multiple orders with the same order date.

You might be tempted to use this solution, which includes a self-contained subquery similar to the one used to return orders on the last actual order date of the month:

```
SELECT Order ID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders
WHERE OrderDate IN
  (SELECT MAX(OrderDate) FROM dbo.Orders
   GROUP BY EmployeeID);
```

However, this solution is incorrect. The result set will include the correct orders (the ones with the maximum *OrderDate* for each employee). But you will also get any order for employee A with an *OrderDate* that happens to be the maximum for employee B, even though it's not also the maximum for employee A. This wasn't an issue with the previous problem, as an order date in month A can't be equal to the maximum order date of a different month B.

In our case, the subquery must be correlated to the outer query, matching the inner *EmployeeID* to the one in the outer row:

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

This query generates the correct results, shown in Table 4-4.

Table 4-4: Orders with the Maximum *OrderDate* for Each Employee

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate
11077	RATTC	1	1998-05-06	1998-06-03
11070	LEHMS	2	1998-05-05	1998-06-02
11073	PERIC	2	1998-05-05	1998-06-02
11063	HUNGO	3	1998-04-30	1998-05-28
11076	BONAP	4	1998-05-06	1998-06-03
11043	SPECD	5	1998-04-22	1998-05-20
11045	BOTTM	6	1998-04-23	1998-05-21
11074	SIMOB	7	1998-05-06	1998-06-03
11075	RICSU	8	1998-05-06	1998-06-03
11058	BLAUS	9	1998-04-29	1998-05-27

The output contains one example of multiple orders for an employee, in the case of employee 2. If you want to return only one row for each employee, you have to introduce a tiebreaker. For example, out of the multiple rows with the maximum *OrderDate*, return the one with the maximum *OrderID*. This can be achieved by adding another subquery that keeps the order only if *OrderID* is equal to the maximum among the orders with the same *EmployeeID* and *OrderDate* as in the outer row:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderDate =
  (SELECT MAX(OrderDate)
   FROM dbo.Orders AS O2
   WHERE O2.EmployeeID = O1.EmployeeID)
  AND OrderID =
  (SELECT MAX(OrderID)
   FROM dbo.Orders AS O2
```

```
  WHERE O2.EmployeeID = O1.EmployeeID
    AND O2.OrderDate = O1.OrderDate);
```

Notice in the output shown in Table 4-5, that of the two orders for employee 2 in the previous query's output, only the one with the maximum *OrderID* remains.

Table 4-5: Most Recent Order for Each Employee; Tiebreaker: *MAX (OrderID)*

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate
11077	RATTC	1	1998-05-06	1998-06-03
11073	PERIC	2	1998-05-05	1998-06-02
11063	HUNGO	3	1998-04-30	1998-05-28
11076	BONAP	4	1998-05-06	1998-06-03
11043	SPECD	5	1998-04-22	1998-05-20
11045	BOTTM	6	1998-04-23	1998-05-21
11074	SIMOB	7	1998-05-06	1998-06-03
11075	RICSU	8	1998-05-06	1998-06-03
11058	BLAUS	9	1998-04-29	1998-05-27

Instead of using two separate subqueries for the sort column (*OrderDate*) and the tiebreaker (*OrderID*), you can use nested subqueries:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderID =
    (SELECT MAX(OrderID)
      FROM dbo.Orders AS O2
      WHERE O2.EmployeeID = O1.EmployeeID
      AND O2.OrderDate =
         (SELECT MAX(OrderDate)
         FROM dbo.Orders AS O3
         WHERE O3.EmployeeID = O1.EmployeeID));
```

I compared the performance of the two and found it very similar, with a slight advantage to the latter. I find the nested approach more complex, so as long as there's no compelling performance benefit, I'd rather stick to the simpler approach. Simpler is easier to understand and maintain, and therefore less prone to errors.

Going back to the simpler approach, for each tiebreaker attribute you have, you need to add a subquery. Each such subquery must be correlated by the group column, sort column, and all preceding tiebreaker attributes. So, to use *MAX (RequiredDate), MAX (OrderID)* as the tiebreaker, you would write the following query:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate
FROM dbo.Orders AS O1
WHERE OrderDate =
  (SELECT MAX(OrderDate)
   FROM dbo.Orders AS O2
   WHERE O2.EmployeeID = O1.EmployeeID)
  AND RequiredDate =
  (SELECT MAX(RequiredDate)
   FROM dbo.Orders AS O2
   WHERE O2.EmployeeID = O1.EmployeeID
     AND O2.OrderDate = O1.OrderDate)

  AND OrderID =
  (SELECT MAX(OrderID)
   FROM dbo.Orders AS O2
   WHERE O2.EmployeeID = O1.EmployeeID
     AND O2.OrderDate = O1.OrderDate
     AND O2.RequiredDate = O1.RequiredDate);
```

The indexing guideline for the tiebreaker queries above is to create an index on (*group_cols, sort_cols*, *tiebreaker_cols*). For example, when the tiebreaker is *MAX(OrderID)*, you want an index on (*EmployeeID*, *OrderDate*, *OrderID*). When the

tiebreaker is *MAX(RequiredDate), MAX(OrderID)*, you want an index on (*EmployeeID*, *OrderDate*, *RequiredDate*, *OrderID*). Such an index would allow retrieving the relevant sort value or tiebreaker value for an employee using a seek operation within the index.

When you're done testing the tiebreaker solutions, run the following code to drop the indexes that were created just for these examples:

```
DROP INDEX dbo.Orders.idx_eid_od_oid;
DROP INDEX dbo.Orders.idx_eid_od_rd_oid;
```

I presented here only one approach to solving tiebreaker problems using ANSI correlated subqueries. This approach is neither the most efficient nor the simplest. You will find other solutions to tiebreaker problems in Chapter 6 in the "Tiebreakers" section, and in Chapter 7 in the "TOP *n* for Each Group" section.

EXISTS

EXISTS is a powerful predicate that allows you to efficiently check whether or not any rows result from a given query. The input to EXISTS is a subquery, which is typically but not necessarily correlated, and the predicate returns TRUE or FALSE, depending on whether the subquery returns at least one row or none. Unlike other predicates and logical expressions, EXISTS cannot return UNKNOWN. Either the input subquery returns rows or it doesn't. If the subquery's filter returns UNKNOWN for a certain row, the row is not returned. Remember that in a filter, UNKNOWN is treated like FALSE. In other words, when the input subquery has a filter, EXISTS will return TRUE only if the filter is TRUE for at least one row. The reason I'm stressing this subtle point will become apparent shortly.

First, let's look at an example that will demonstrate the use of EXISTS. The following query returns all customers from Spain that made orders, generating the output shown in Table 4-6:

Table 4-6: Customers from Spain that Made Orders

CustomerID	CompanyName
BOLID	Bólido Comidas preparadas
GALED	Galería del gastrónomo
GODOS	Godos Cocina Típica
ROMEY	Romero y tomillo

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
  AND EXISTS
    (SELECT* FROM Orders AS O
     WHERE O.CustomerID = C.CustomerID);
```

The outer query returns customers from Spain for whom the EXISTS predicate finds at least one order row in the Orders table with the same *CustomerID* as in the outer customer row.

> **Tip** The use of * here is perfectly safe, even though in general it's not a good practice. The optimizer ignores the SELECT list specified in the subquery because EXISTS cares only about the existence of rows and not about any specific attributes.

Examine the execution plan produced for this query, as shown in Figure 4-1.

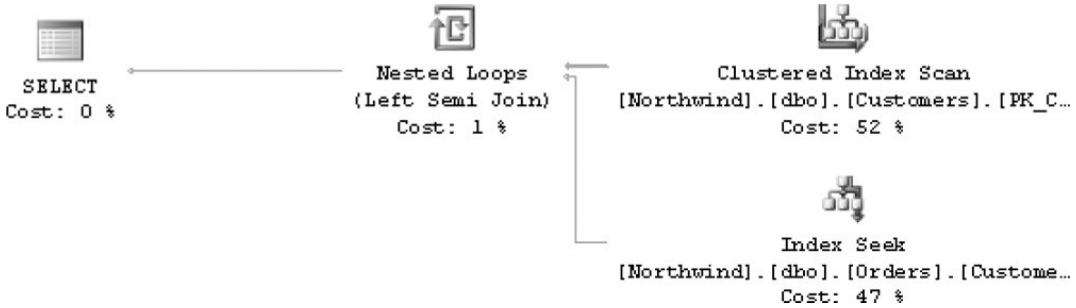

Figure 4-1: Execution plan for an EXISTS query

The plan scans the Customers table and filters customers from Spain. For each matching customer, the plan performs a seek within the index on *Orders.CustomerID* to check whether the Orders table contains an order with that customer's *CustomerID*. The index on the filtered column in the subquery (*Orders.CustomerID* in our case) is very helpful here, because it provides direct access to the rows of the Orders table with a given *CustomerID* value.

EXISTS vs. IN Programmers frequently wonder whether a query with the EXISTS predicate is more efficient than a logically equivalent query with the IN predicate. For example, the last query could be written using an IN predicate with a self-contained subquery as follows:

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
  AND CustomerID IN(SELECT CustomerID FROM dbo. Orders);
```

In versions prior to SQL Server 2000, I used to see differences between plans generated for EXISTS and for IN, and typically EXISTS performed better because of its inherent short-circuiting. However, in SQL Server 2000 and 2005 the optimizer usually generates identical plans for the two queries *when they are truly logically equivalent*, and this case qualifies.

The plan generated for the last query using IN is identical to the one shown in Figure 4-1, which was generated for the query using EXISTS.

If you're always thinking of the implications of three-valued logic, you might realize that there is a difference between IN and EXISTS. Unlike EXISTS, IN can in fact produce an UNKNOWN logical result when the input list contains a NULL. For example, *a IN(b, c, NULL)* is UNKNOWN. However, because UNKNOWN is treated like FALSE in a filter, the result of a query with the IN predicate is the same as with the EXISTS predicate, and the optimizer is aware of that, hence the identical plans.

NOT EXISTS vs. NOT IN The logical difference between EXISTS and IN does show up if we compare NOT EXISTS and NOT IN, when the input list of NOT IN might contain a NULL.

For example, suppose you need to return customers from Spain who made no orders. Here's the solution using the NOT EXISTS predicate, which generates the output shown in Table 4-7:

Table 4-7: Customers from Spain Who Made No Orders

CustomerID	CompanyName
FISSA	FISSA Fabrica Inter. Salchichas S.A.

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
  AND NOT EXISTS
    (SELECT * FROM Orders AS O
     WHERE O.CustomerID = C.CustomerID);
```

Even if there is a NULL *CustomerID* in the Orders table, it is of no concern to us. You get all customers from Spain for which SQL Server cannot find even one row in the Orders table with the same *CustomerID*. The plan generated for this query is shown in Figure 4-2.

Figure 4-2: Execution plan for a NOT EXISTS query

The plan scans the Customers table and filters customers from Spain. For each matching customer, the plan performs a

seek within the index on *Orders.CustomerID*. The *Top* operator appears because it's only necessary to see whether there's at least one matching order for the customer—that's the short-circuiting capability of EXISTS in action. This use of *Top* is particularly efficient when the *Orders.CustomerID* column has a high density (that is, a large number of duplicates). The seek takes place only once for each customer, and regardless of the number of orders the customer has, only one row is scanned at the leaf level (bottom level of the index) to look for a match, as opposed to all matching rows.

In this case, the following solution using the NOT IN predicate does yield the same output. It seems to have the same meaning, but we'll see later that it does not.

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
  AND CustomerID NOT IN(SELECT CustomerID FROM dbo.Orders);
```

If you examine the execution plan, shown in Figure 4-3, you will find that it's different from the one generated for the NOT EXISTS query.

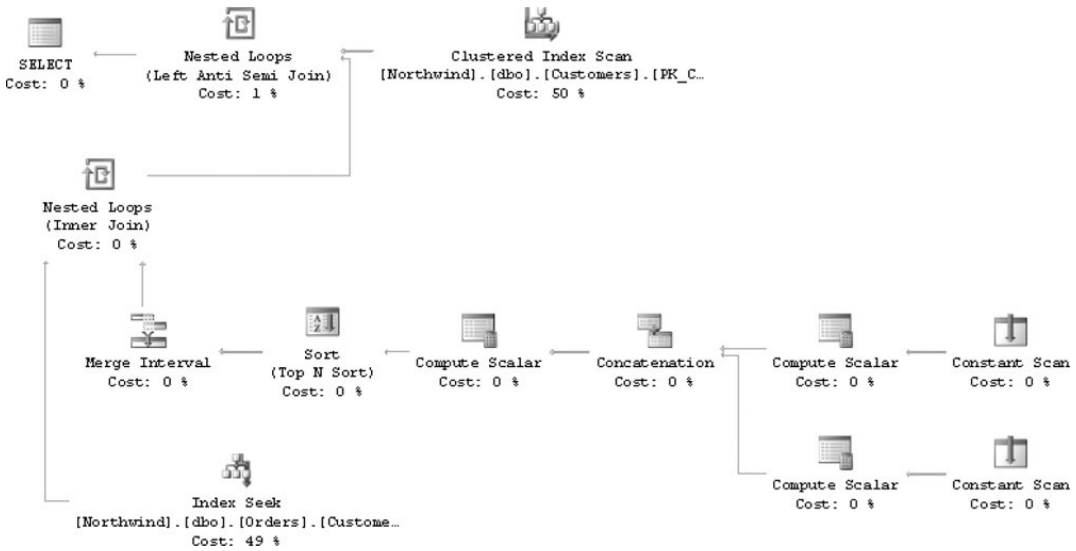

Figure 4-3: Execution plan for a NOT IN query

There are some additional operations at the beginning of this plan, compared to the previous plan—steps needed to look for NULL *CustomerID* s. Why is this plan different than the one generated for the NOT EXISTS query? And why would SQL Server care particularly about the existence of NULLs in *Orders.CustomerID*?

The discrepancy between the plans doesn't affect the result because there's no row in the Orders table with a NULL *CustomerID*. However, because the *CustomerID* column allows NULLs, the optimizer must take this fact into consideration. Let's see what happens if we add a row with a NULL *CustomerID* to the Orders table:

```
INSERT INTO dbo.Orders DEFAULT VALUES;
```

Now rerun both the NOT EXISTS and NOT IN queries. You will find that the NOT EXISTS query still returns the same output as before, while the NOT IN query now returns an empty set. In fact, when there's a NULL in the *Orders.CustomerID* column, the NOT IN query will always return an empty set. The reason is that the predicate *val IN (val1, val2, …, NULL)* can never return FALSE; rather, it can return only TRUE or UNKNOWN. As a result, *val* ***NOT*** *IN (val1, val2, …, NULL)* can only return NOT TRUE or NOT UNKNOWN, neither of which is TRUE.

For example, suppose the customer list in this query is *(a, b, NULL)*. Customer *a* appears in the list, and therefore the predicate *a IN(a, b, NULL)* returns TRUE. The predicate *a NOT IN(a, b, NULL)* returns NOT TRUE, or FALSE, and customer *a* is not returned by the query. Customer *c*, on the other hand, does *not* appear in the list *(a, b, NULL)*, but the logical result of *c IN(a, b, NULL)* is UNKNOWN, because of the NULL. The predicate *b NOT IN(a, b, NULL)* therefore returns NOT UNKNOWN, which equals UNKNOWN, and customer *c* is not returned by the query, either, even though *c* does not appear in the customer list. Whether or not a customer appears in the customer list, if the list contains NULL, the customer is not returned by the query. You realize that when NULLs are potentially involved (such as when the queried column allows NULLs), NOT EXISTS and NOT IN are not logically equivalent. This explains the discrepancy between the plans and the potential difference in results. To make the NOT IN query logically equivalent to the EXISTS query, declare

the column as NOT NULL (if appropriate) or add a filter to the subquery to exclude NULLs:

```
SELECT CustomerID, CompanyName
FROM dbo.Customers AS C
WHERE Country = N'Spain'
  AND CustomerID NOT IN(SELECT CustomerID FROM dbo.Orders
                        WHERE CustomerID IS NO TNULL);
```

This query generates the same result as the NOT EXISTS query, as well as the same plan.

Once you're done testing the queries, make sure you remove the row with the NULL *CustomerID*:

```
DELETE FROM dbo.Orders WHERE CustomerID IS NULL;
```

Minimum Missing Value To put your knowledge of the EXISTS predicate into action, try to solve the following problem. First create and populate the table T1 by running the code in Listing 4-1.

Listing 4-1: Creating and populating the table T1

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

CREATE TABLE dbo.T1
(
  keycol  INT         NOT NULL PRIMARY KEY CHECK(keycol > 0),
  datacol VARCHAR(10) NOT NULL
);
INSERT INTO dbo.T1(keycol, datacol) VALUES(3, 'a');
INSERT INTO dbo.T1(keycol, datacol) VALUES(4, 'b');
INSERT INTO dbo.T1(keycol, datacol) VALUES(6, 'c');
INSERT INTO dbo.T1(keycol, datacol) VALUES(7, 'd');
```

Notice that *keycol* must be positive. Your task is to write a query that returns the lowest missing key, assuming that key values start at 1. For example, the table is currently populated with the keys 3, 4, 6, and 7, so your query should return the value 1. If you insert two more rows, with the keys 1 and 2, your query should return 5.

Solution:

Here's a suggested CASE expression (incomplete) that I used in my solution:

```
SELECT
  CASE
    WHEN NOT EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1) THEN 1
    ELSE (...subquery returning minimum missing value...)
  END;
```

If 1 doesn't exist in the table, the CASE expression returns 1; otherwise, it returns the result of a subquery returning the minimum missing value.

Here's the subquery that I used to return the minimum missing value:

```
SELECT MIN(A.keycol + 1) as missing
FROM dbo.T1 AS A
WHERE NOT EXISTS
  (SELECT * FROM dbo.T1 AS B
   WHERE B.keycol = A.keycol + 1);
```

The NOT EXISTS predicate returns TRUE only for values in T1 that are right before a gap (4 and 7 in our case). A value is right before a gap if the value plus one does not exist in the same table. The outer T1 table has the alias A, and the inner T1 table has the alias B. You could use the expression *B.keycol–1 = A.keycol* in the subquery's filter; although it might be a bit confusing to use such an expression when looking for a value in B that is greater than the value in A by one. If you think about it, for *B.keycol* to be greater than *A.keycol* by one, *B.keycol* minus one must be equal to *A.keycol*. If this logic confuses you, you can use *B.keycol = A.keycol + 1* instead, as I did. Once all points before gaps are isolated, the outer query returns the minimum plus one, which is the first missing value in the first gap. Make a mental note of the technique to identify a point before a gap, as it's a very handy key technique.

Now you can incorporate the query returning the minimum missing value in the CASE expression:

```
SELECT
  CASE
    WHEN NOT EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1) THEN 1
    ELSE (SELECT MIN(keycol + 1)
          FROM dbo.T1 AS A
          WHERE NOT EXISTS
            (SELECT *
             FROM dbo.T1 AS B
             WHERE B.keycol = A.keycol + 1))
  END;
```

If you run this query with the sample data inserted by Listing 4-1, you should get 1 as the result. If you then insert two more rows, with the keys 1 and 2 (as shown in the following code), and rerun the query, you should get 5 as the result.

```
INSERT INTO dbo.T1(keycol, datacol) VALUES(1, 'e');
INSERT INTO dbo.T1(keycol, datacol) VALUES(2, 'f');
```

Here is an example of how you might use the CASE expression for the minimum missing key in an INSERT … SELECT statement, perhaps in a scenario where you needed to reuse deleted keys:

```
INSERT INTO dbo.T1(keycol, datacol)
  SELECT
    CASE
      WHEN NOT EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1) THEN 1
      ELSE (SELECT MIN(keycol + 1)
            FROM dbo.T1 AS A
            WHERE NOT EXISTS
              (SELECT *
              FROM dbo.T1 AS B
              WHERE B.keycol = A.keycol + 1))
    END,
    'f';
```

Query the T1 table after running this INSERT, and notice in the output shown in Table 4-8 that the insert generated the key value 5, which was the minimum missing key:

Table 4-8: Content of T1 after Insert

keycol	datacol
1	e
2	f
3	a
4	b
5	f
6	c
7	d

```
SELECT * FROM dbo.T1;
```

Note Multiple processes running such code simultaneously might get the same key. You can overcome this issue by introducing error-handling code that traps a duplicate key error and then retries. There are other more efficient techniques to reuse deleted keys, but they are more complex and require you to maintain a table with ranges of missing values. Also note that reusing deleted keys is not often a good idea, for reasons beyond concurrency. Here I just wanted to give you a chance to practice with the EXISTS predicate.

Note that you can merge the two cases where 1 does exist in the table, and where 1 doesn't, instead of using a CASE expression. The solution requires some tricky logical manipulation:

```
SELECT COALESCE(MIN(keycol + 1), 1)
FROM dbo.T1 AS A
WHERE NOT EXISTS
```

```
  (SELECT * FROM dbo.T1 AS B
   WHERE B.keycol = A.keycol + 1)
  AND EXISTS(SELECT * FROM dbo.T1 WHERE keycol = 1);
```

The query has both logical expressions from the CASE expression in the WHERE clause. It returns the minimum missing value if 1 does exist in the table (that is, when the second EXISTS predicate is always TRUE). If 1 doesn't exist in the table (that is, the second EXISTS predicate is always FALSE), the filter generates an empty set and the expression *MIN(keycol) + 1* yields a NULL. The value of the COALESCE expression is then 1.

Even though this solution achieves the request with a single query, I personally like the original solution better. This solution is a bit tricky and isn't as intuitive as the previous one, and simplicity and readability of code goes a long way.

Reverse Logic Applied to Relational Division Problems Our minds are usually accustomed to think in positive terms. However, positive thinking in some cases can get you only so far. In many fields, including SQL programming, negative thinking or reverse logic can give you new insight or be used as another tool to solve problems. Applying reverse logic can in some cases lead to simpler or more efficient solutions than applying a positive approach. It's another tool in your toolbox.

Euclid, for example, was very fond of applying reverse logic in his mathematical proofs (proof by way of negation). He used reverse logic to prove that there's an infinite number of prime numbers. By contradicting a certain assumption, creating a paradox, you prove that the opposite must be true.

Before I demonstrate an application of reverse logic in SQL, I'd like to deliver the idea through an ancient puzzle. Two guards stand in front of two doors. One door leads to gold and treasures, and the other leads to sudden death, but you don't know which is which. One of the guards always tells the truth and the other always lies, but you don't know who the liar is and who's the sincere one (even though the guards do). Obviously, you want to enter the door that leads to the gold and not to sudden death. You have but one opportunity to ask one of the guards a question; what will the question be?

Any question that you ask applying positive thinking will not give you 100-percent assurance of picking the door that leads to the gold. However, applying reverse logic can give you that assurance.

Ask either guard, "If I ask the other guard where the door is that leads to the gold, which door would he point to?" If you asked the sincere guard, he would point at the door that leads to sudden death, knowing that the other is a liar. If you asked the liar, he'd also point at the door that leads to sudden death, knowing that the other guard is sincere and would point to the door that leads to the gold. All you would have to do is enter the door that was not pointed at.

Reverse logic is sometimes a handy tool in solving problems with SQL. An example of where you can apply reverse logic is in solving relational division problems. At the beginning of the chapter, I discussed the following problem: from the Northwind database, return all customers with orders handled by all employees from the USA. I showed an example to solving the problem that used positive thinking. To apply reverse logic, you first need to be able to phrase the request in a negative way. Instead of saying, "Return customers for whom all USA employees handled orders," you can say, "Return customers for whom no USA employee handled no order." Remember that two negatives produce a positive. If for customer A you cannot find even one USA employee who did not handle any orders, then all USA employees must have handled orders for customer A.

Once you phrase the request in a negative way, the translation to SQL is intuitive using correlated subqueries:

```
SELECT * FROM dbo.Customers AS C
WHERE NOT EXISTS
  (SELECT * FROM dbo.Employees AS E
   WHERE Country = N'USA'
     AND NOT EXISTS
       (SELECT * FROM dbo.Orders AS O
        WHERE O.CustomerID = C.CustomerID
          AND O.EmployeeID = E.EmployeeID));
```

When you "read" the query, it really sounds like the English phrasing of the request:

```
Return customers
for whom you cannot find
  any employee
  from the USA
  for whom you cannot find
    any order
    placed for the subject customer
    and by the subject employee
```

You get the same 23 customers back as the ones shown in Table 4-1 returned by the query applying the positive approach. Notice, though, that the negative solution gives you access to all the customer attributes, while the positive solution gives you access only to the customer IDs. To access other customer attributes, you need to add a join between the result set and the Customers table.

When comparing the performance of the solutions in this case, the solution applying the positive approach performs better. In other cases, the negative approach might yield better performance. You now have another tool that you can use when solving problems.

Misbehaving Subqueries

There's a very tricky programming error involving subqueries that I've seen occasionally and have even had the misfortune to introduce into production code myself. I'll first describe the bug, and then make recommendations on what you can do to avoid it.

Suppose that you are asked to return the shippers from the Northwind database that did not ship orders to customer LAZYK. Examining the data, shipper 1 (Speedy Express) is the only one that qualifies. The following query is supposed to return the desired result:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers
WHERE ShipperID NOT IN
  (SELECT ShipperID FROM dbo.Orders
   WHERE CustomerID = N'LAZYK');
```

Surprisingly, this query returns an empty set. Can you tell why? Can you identify the elusive bug in my code?

Well, apparently the column in the Orders table holding the *ShipperID* is called *ShipVia* and not *ShipperID*. There is no *ShipperID* column in the Orders table. Realizing this, you'd probably expect the query to have failed because of the invalid column name. Sure enough, if you run only the part that was supposed to be a self-contained subquery, it does fail: Invalid column name 'ShipperID'. However, in the context of the outer query, apparently the subquery is valid! The name resolution process works from the inner nesting level outward. The query processor first looked for a *ShipperID* column in the Orders table, which is referenced in the current level. Not having found such a column name, it looked for one in the Shippers table—the outer level—and found it. Unintentionally, the subquery became correlated, as if it were written as the following illustrative code:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers AS S
WHERE ShipperID NOT IN
  (SELECT S.ShipperID FROM dbo.Orders AS O
   WHERE O.CustomerID = N'LAZYK');
```

Logically, the query doesn't make much sense of course; nevertheless, it is technically valid.

You can now understand why you got an empty set back. Unless there's no order for customer LAZYK anywhere in the Orders table, obviously shipper *n* is always going to be in the set (*SELECT n FROM dbo.Orders WHERE CustomerID = 'LAZYK'*). And the NOT IN predicate will always yield FALSE. This buggy query logically became a nonexistence query equivalent to the following illustrative code:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers
WHERE NOT EXISTS
  (SELECT * FROM dbo.Orders
   WHERE CustomerID = N'LAZYK');
```

To fix the problem, of course, you should use the correct name for the column from Orders that holds the *ShipperID*—*ShipVia*:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers AS S
WHERE ShipperID NOT IN
  (SELECT Ship Via FROM dbo.Orders AS O
   WHERE CustomerID = N'LAZYK');
```

This will generate the expected result shown in Table 4-9.

Table 4-9: Shippers that Did

Not Ship Orders to Customer LAZYK

ShipperID	CompanyName
1	Speedy Express

However, to avoid such bugs in the future, it's a good practice to always include the table name or alias for all attributes in a subquery, even when the subquery is self-contained. Had I aliased the *ShipperID* column in the subquery (as shown in the following code), a name resolution error would have been generated and the bug would have been detected:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers AS S
WHERE ShipperID NOT IN
  (SELECT O.ShipperID FROM dbo.Orders AS O
   WHERE O.CustomerID = N'LAZYK');

Msg 207, Level 16, State 1, Line 4
Invalid column name 'ShipperID'.
```

Finally, correcting the bug, here's how the solution query should look:

```
SELECT ShipperID, CompanyName
FROM dbo.Shippers AS S
WHERE ShipperID NOT IN
  (SELECT O.ShipVia FROM dbo.Orders AS O
   WHERE O.CustomerID = N'LAZYK');
```

Uncommon Predicates

In addition to IN and EXISTS, there are three more predicates in SQL, but they are rarely used: ANY, SOME, and ALL. You can consider them to be generalizations of the IN predicate. (ANY and SOME are synonyms, and there is no logical difference between them.)

An IN predicate is translated to a series of equality predicates separated by OR operators—for example, *v IN(x, y, z)* is translated to *v = x OR v = y OR v = z*. ANY (or SOME) allows you to specify the comparison you want in each predicate, not limiting you to the equality operator. For example, *v <> ANY(x, y, z)* is translated to *v <> x OR v <> y OR v <> z.*

ALL is similar, but it's translated to a series of logical expressions separated by AND operators. For example, *v <> ALL(x, y, z)* is translated to *v <> x AND v <> y AND v <> z.*

> **Note** IN allows as input either a list of literals or a subquery returning a single column. ANY/SOME and ALL support only a subquery as input. If you have the need to use these uncommon predicates with a list of literals as input, you must convert the list to a subquery. So, instead of *v <> ANY(x, y, z)*, you would use *v <> ANY(SELECT x UNION ALL SELECT y UNION ALL SELECT z)*.

To demonstrate the use of these uncommon predicates, let's suppose you were asked to return, for each employee, the order with the minimum *OrderID.* Here's how you can achieve this with the ANY operator, which would generate the result shown in Table 4-10:

Table 4-10: Row with the Minimum *OrderID* for Each Employee

OrderID	CustomerID	EmployeeID	OrderDate
10248	VINET	5	1996-07-04
10249	TOMSP	6	1996-07-05
10250	HANAR	4	1996-07-08
10251	VICTE	3	1996-07-08
10255	RICSU	9	1996-07-12
10258	ERNSH	1	1996-07-17
10262	RATTC	8	1996-07-22
10265	BLONP	2	1996-07-25

10289	BSBEV	7	1996-08-26

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders AS O1
WHERE NOT OrderID >
  ANY(SELECT OrderID
      FROM dbo.Orders AS O2
      WHERE O2.EmployeeID = O1.EmployeeID);
```

A row has the minimum *OrderID* for an employee if its *OrderID* is not greater than any *OrderID* for the same employee.

You can also write a query using ALL to achieve the same thing:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders AS O1
WHERE OrderID <=
  ALL(SELECT OrderID
      FROM dbo.Orders AS O2
      WHERE O2.EmployeeID = O1.EmployeeID);
```

A row has the minimum *OrderID* for an employee if its *OrderID* is less than or equal to all *OrderID* s for the same employee.

None of the solutions above would fall into the category of intuitive solutions, and maybe this can explain why these predicates are not commonly used. The natural way to write the solution query would probably be as follows:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate
FROM dbo.Orders AS O1
WHERE OrderID =
  (SELECT MIN(OrderID)
   FROM dbo.Orders AS O2
   WHERE O2.EmployeeID = O1.EmployeeID);
```

Table Expressions

So far, I've covered scalar and multi-valued subqueries. This section deals with table subqueries, which are known as *Table Expressions.* In this chapter, I'll discuss derived tables and the new common table expressions (CTE).

More Info For information about the two other types of table expressions—views and user-defined functions (UDF)—please refer to *Inside Microsoft SQL Server 2005: T-SQL Programming* (Microsoft Press, 2006).

Derived Tables

A derived table is a table expression—that is, a virtual result table derived from a query expression. A derived table appears in the FROM clause of a query like any other table. The scope of existence of a derived table is the outer query's scope only.

The general form in which a derived table is used is as follows:

```
FROM (derived_table_query expression) AS derived_table_alias
```

Note A derived table is completely virtual. It's not physically materialized, nor does the optimizer generate a separate plan for it. The outer query and the inner one are merged, and one plan is generated. You shouldn't have any special concerns regarding performance when using derived tables. Merely using derived tables neither degrades nor improves performance. Their use is more a matter of simplification and clarity of code.

A derived table must be a valid table; therefore, it must follow several rules:

- All columns must have names.
- The column names must be unique.
- ORDER BY is not allowed (unless TOP is also specified).

Note Unlike scalar and multivalued subqueries, derived tables cannot be correlated; they must be self-contained. There's an exception to this rule when using the new APPLY operator, which I'll cover in Chapter 7.

Result Column Aliases

One of the uses of derived tables is to enable the reuse of column aliases when expressions are so long you'd rather not repeat them. For simplicity's sake, I'll demonstrate column alias reuse with short expressions.

Remember from Chapter 1 that aliases created in the query's SELECT list cannot be used in most of the query elements. The reason for this is that the SELECT clause is logically processed almost last, just before the ORDER BY clause. For this reason, the following illustrative query fails:

```
SELECT
  YEAR(OrderDate) AS OrderYear,
  COUNT(DISTINCT CustomerID) AS NumCusts
FROM dbo.Orders
GROUP BY OrderYear;
```

The GROUP BY clause is logically processed before the SELECT clause, so at the GROUP BY phase, the *OrderYear* alias has not yet been created.

By using a derived table that contains only the SELECT and FROM elements of the original query, you can create aliases and make them available to the outer query in any element.

There are two formats of aliasing the derived table's result columns. One is inline column aliasing:

```
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
      FROM dbo.Orders) AS D
GROUP BY OrderYear;
```

And the other is external column aliasing following the derived table's alias:

```
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM (SELECT YEAR(OrderDate), CustomerID
      FROM dbo.Orders) AS D(OrderYear, CustomerID)
GROUP BY OrderYear;
```

These days, I typically use inline column aliasing, as I find it much more practical. You don't have to specify aliases for base columns, and it's much more convenient to troubleshoot. When you highlight and run only the derived table query, the result set you get includes all result column names. Also, it's clear which column alias belongs to which expression.

The external column aliasing format lacks all the aforementioned benefits. I can't think of even one advantage it has. Because external column aliasing is not common knowledge among SQL programmers, you might find it kind of cool, as using it could demonstrate a higher level of mastery of SQL. I know I did at some stage. Of course, it's not as noble a guideline as clarity of code and ease of troubleshooting and maintenance.

Using Arguments

Even though a derived table query cannot be correlated, it can refer to variables defined in the same batch. For example, the following code returns for each year the number of customers handled by employee 3, and it generates the output shown in Table 4-11:

Table 4-11: Yearly Count of Customers Handled by Employee 3

OrderYear	NumCusts
1996	16
1997	46
1998	30

```
DECLARE @EmpID AS INT;
SET @EmpID = 3;

SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
      FROM dbo.Orders
      WHERE EmployeeID = @EmpID) AS D
GROUP BY OrderYear;
```

Nesting

Derived tables can be nested. Logical processing in a case of nested derived tables starts at the innermost level and proceeds outward.

The following query (which produces the output shown in Table 4-12) returns the order year and the number of customers for years with more than 70 active customers:

Table 4-12: Order Year and Number of Customers for Years with More than 70 Active Customers

OrderYear	NumCusts
1997	86
1998	81

```
SELECT OrderYear, NumCusts
FROM (SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
      FROM (SELECT YEAR(OrderDate) AS OrderYear, CustomerID
            FROM dbo.Orders) AS D1
      GROUP BY OrderYear) AS D2
WHERE NumCusts > 70;
```

Multiple References

Out of all the types of table expressions available in T-SQL, derived tables are the only type that suffers from a certain limitation related to multiple references. You can't refer to the same derived table multiple times in the same query. For example, suppose you want to compare each year's number of active customers to the previous year's. You want to join two instances of a derived table that contains the yearly aggregates. In such a case, unfortunately, you have to create two derived tables, each repeating the same derived table query:

```
SELECT Cur.OrderYear,
  Cur.NumCusts AS CurNumCusts, Prv.NumCusts AS PrvNumCusts,
  Cur.NumCusts - Prv.NumCusts AS Growth
FROM (SELECT YEAR(OrderDate) AS OrderYear,
        COUNT(DISTINCT CustomerID) AS NumCusts
      FROM dbo.Orders
      GROUP BY YEAR(OrderDate)) AS Cur
  LEFT OUTER JOIN
     (SELECT YEAR(OrderDate) AS OrderYear,
        COUNT(DISTINCT CustomerID) AS NumCusts
      FROM dbo.Orders
      GROUP BY YEAR(OrderDate)) AS Prv
    ON Cur.OrderYear = Prv.OrderYear + 1;
```

The output of this query is shown in Table 4-13.

Table 4-13: Comparing Current Year to Previous Year's Number of Customers

OrderYear	CurNumCusts	PrvNumCusts	Growth
1996	67	NULL	NULL
1997	86	67	19
1998	81	86	−5

Common Table Expressions (CTE)

A common table expression (CTE) is a new type of table expression introduced in SQL Server 2005. As for the standard, CTEs were introduced in the ANSI SQL: 1999 specification. In many aspects, you will find CTEs very similar to derived tables. However, CTEs have several important advantages, which I'll describe in this section.

Remember that a derived table appears in its entirety in the FROM clause of an outer query. A CTE, however, is defined first using a WITH statement, and an outer query referring to the CTE's name follows the CTE's definition:

```
WITH cte_name
AS
(
  cte_query
)
outer_query_referringto_cte_name;
```

Note Because the WITH keyword is used in T-SQL for other purposes as well, to avoid ambiguity, the statement preceding the CTE's WITH clause must be terminated with a semicolon. The use of a semicolon to terminate statements is supported by ANSI. It's a good practice, and you should start getting used to it even where T-SQL currently doesn't require it.

A CTE's scope of existence is the outer query's scope. It's not visible to other statements in the same batch.

The same rules I mentioned for the validity of a derived table's query expression apply to the CTE's as well. That is, the query must generate a valid table, so all columns must have names, all column names must be unique, and ORDER BY is not allowed (unless TOP is also specified).

Next, I'll go over aspects of CTEs, demonstrating their syntax and capabilities, and compare them to derived tables.

Result Column Aliases

Just as you can with derived tables, you can provide aliases to result columns either inline in the CTE's query or externally in parentheses following the CTE's name. The following code illustrates the first method:

```
WITHC AS
(
  SELECT YEAR(OrderDate) AS OrderYear, CustomerID
  FROM dbo.Orders
)
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM C
GROUP BY OrderYear;
```

The next bit of code illustrates how to provide aliases externally in parentheses following the CTE's name:

```
WITH C(OrderYear, CustomerID) AS
(
  SELECT YEAR(OrderDate), CustomerID
  FROM dbo.Orders
)
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM C
GROUP BY OrderYear;
```

Using Arguments

Another similarity between CTEs and derived tables is that CTEs can refer to variables declared in the same batch:

```
DECLARE @EmpID AS INT;
SET @EmpID = 3;

WITH C AS
(
  SELECT YEAR(OrderDate) AS OrderYear, CustomerID
  FROM dbo.Orders
  WHERE EmployeeID = @EmpID
)
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
FROM C
GROUP BY OrderYear;
```

Multiple CTEs

Unlike derived tables, CTEs cannot be nested directly. That is, you cannot define a CTE within another CTE. However, you can define multiple CTEs using the same WITH statement, each of which can refer to the preceding CTEs. The outer query has access to all the CTEs. Using this capability, you can achieve the same result you would by nesting derived tables. For example, the following WITH statement defines two CTEs:

```
WITH C1 AS
```

```
(
  SELECT YEAR(OrderDate) AS OrderYear, CustomerID
  FROM dbo.Orders
),
C2 AS
(
  SELECT OrderYear, COUNT(DISTINCT CustomerID) AS NumCusts
  FROM C1
  GROUP BY OrderYear
)
SELECT OrderYear, NumCusts
FROM C2
WHERE NumCusts > 70;
```

C1 returns order years and customer IDs for each order, generating the *OrderYear* alias for the order year. C2 groups the rows returned from C1 by *OrderYear* and calculates the count of distinct *CustomerID* s (number of active customers). Finally, the outer query returns only order years with more than 70 active customers.

Multiple References

One of the advantages CTEs have over derived tables is that you can refer to the same CTE name multiple times in the outer query. You don't need to repeat the same CTE definition like you do with derived tables. For example, the following code demonstrates a CTE solution for the request to compare each year's number of active customers to the previous year's number:

```
WITH YearlyCount AS
(
  SELECT YEAR(OrderDate) AS OrderYear,
    COUNT(DISTINCT CustomerID) AS NumCusts
  FROM dbo.Orders
  GROUP BY YEAR(OrderDate)
)

SELECT Cur.OrderYear,
  Cur.NumCusts AS CurNumCusts, Prv.NumCusts AS PrvNumCusts,
  Cur.NumCusts - Prv.NumCusts AS Growth
FROM YearlyCount AS Cur
  LEFT OUTER JOIN YearlyCount AS Prv
    ON Cur.OrderYear = Prv.OrderYear + 1;
```

You can see that the outer query refers to the *YearlyCount* CTE twice—once representing the current year (*Cur*), and once representing the previous year (*Prv*).

Modifying Data

You can modify data through CTEs. To demonstrate this capability, first run the code in Listing 4-2 to create and populate the dbo.CustomersDups table with sample data.

Listing 4-2: Creating and populating the CustomersDups table

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

WITH CrossCustomers AS
(
  SELECT 1 AS c, C1.*
  FROM dbo.Customers AS C1, dbo.Customers AS C2
)
SELECT ROW_NUMBER() OVER(ORDER BY c) AS KeyCol,
  CustomerID, CompanyName, ContactName, ContactTitle, Address,
  City, Region, PostalCode, Country, Phone, Fax
INTO dbo.CustomersDups
FROM CrossCustomers;

CREATE UNIQUE INDEX idx_CustomerID_KeyCol
  ON dbo.CustomersDups(CustomerID, KeyCol);
```

Note that I used a new function called ROW_NUMBER here to create sequential integers that will be used as unique keys. In SQL Server 2000, you can achieve this by using the IDENTITY function in the SELECT INTO statement. I'll discuss this function in detail later in this chapter.

Basically, the code in Listing 4-2 creates a table of customers with a lot of duplicate occurrences of each customer. The following code demonstrates how you can remove duplicate customers using a CTE.

```
WITH JustDups AS
(
  SELECT * FROM dbo.CustomersDups AS C1
  WHERE KeyCol <
    (SELECT MAX(KeyCol) FROM dbo.CustomersDups AS C2
     WHERE C2.CustomerID = C1.CustomerID)
)
DELETE FROM JustDups;
```

The CTE *JustDups* has all duplicate rows for each customer, not including the row where *KeyCol* is the maximum for the customer. Notice that the code in Listing 4-2 creates an index on (*CustomerID, KeyCol*) to support the filter. The outer query merely deletes all rows from *JustDups.* After this code is run, the CustomersDups table contains only unique rows. At this point, you can create a primary key or a unique constraint on the *CustomerID* column to avoid duplicates in the future.

Container Objects

CTEs can be used in container objects such as views and inline UDFs. This capability provides encapsulation, which is important for modular programming. Also, I mentioned earlier that CTEs cannot be nested directly. However, you can nest CTEs indirectly by encapsulating a CTE in a container object and querying the container object from an outer CTE.

Using CTEs in views or inline UDFs is very trivial. The following example creates a view returning a yearly count of customers:

```
CREATE VIEW dbo.VYearCnt
AS
WITH YearCnt AS
(
  SELECT YEAR(OrderDate) AS OrderYear,
    COUNT(DISTINCT CustomerID) AS NumCusts
  FROM dbo.Orders
  GROUP BY YEAR(OrderDate)
)
SELECT * FROM YearCnt;
GO
```

Querying the view, as shown in the following code, returns the output shown in Table 4-14:

Table 4-14: Yearly Count of Customers

OrderYear	NumCusts
1996	67
1997	86
1998	81

```
SELECT * FROM dbo.VYearCnt;
```

If you want to pass an input argument to the container object—for example, return the yearly count of customers for the given employee—you'd create an inline UDF as follows:

```
CREATE FUNCTION dbo.fn_EmpYearCnt(@EmpID AS INT) RETURNS TABLE
AS
RETURN
  WITH EmpYearCnt AS
  (
    SELECT YEAR(OrderDate) AS OrderYear,
    COUNT(DISTINCT CustomerID) AS NumCusts

    FROM dbo.Orders
    WHERE EmployeeID = @EmpID
```

```
    GROUP BY YEAR(OrderDate)
  )
  SELECT * FROM EmpYearCnt;
GO
```

Querying the UDF and providing employee ID 3 as input returns the output shown in Table 4-15:

Table 4-15: Yearly Count of Customers

OrderYear	NumCusts
1996	16
1997	46
1998	30

```
SELECT * FROM dbo.fn_EmpYearCnt(3);
```

Recursive CTEs

Recursive CTEs represent one of the most significant T-SQL enhancements in SQL Server 2005. Finally, SQL Server supports recursive querying capabilities with pure set-based queries. The types of tasks and activities that can benefit from recursive queries include manipulation of graphs, trees, hierarchies, and many others. Here I'll just introduce you to recursive CTEs. For more information and detailed applications, you can find extensive coverage in Chapter 9.

I'll describe a recursive CTE using an example. You're given an input *EmployeeID* (for example, employee 5) from the Employees table in the Northwind database. You're supposed to return the input employee and subordinate employees in all levels, based on the hierarchical relationships maintained by the *EmployeeID* and *ReportsTo* attributes. The attributes you need to return for each employee include: *EmployeeID*, *ReportsTo*, *FirstName*, and *LastName.*

Before I demonstrate and explain the recursive CTE's code, I'll create the following covering index, which is optimal for the task:

```
CREATE UNIQUE INDEX idx_mgr_emp_ifname_ilname
  ON dbo.Employees(ReportsTo, EmployeeID)
  INCLUDE(FirstName, LastName);
```

This index will allow fetching direct subordinates of each manager by using a single seek plus a partial scan. Note the included columns (*FirstName* and *LastName*) that were added for covering purposes.

Here's the recursive CTE code that will return the desired result, which is shown in Table 4-16:

Table 4-16: Subordinates of Employee 5

EmployeeID	ReportsTo	FirstName	LastName
5	2	Steven	Buchanan
6	5	Michael	Suyama
7	5	Robert	King
9	5	Anne	Dodsworth

```
WITH EmpsCTE AS
(
  SELECT EmployeeID, ReportsTo, FirstName, LastName
  FROM dbo.Employees
  WHERE EmployeeID = 5

  UNION ALL

  SELECT EMP.EmployeeID, EMP.ReportsTo, EMP.FirstName, EMP.LastName
  FROM EmpsCTE AS MGR
    JOIN dbo.Employees AS EMP
      ON EMP.ReportsTo = MGR.EmployeeID
)
SELECT * FROM EmpsCTE;
```

A recursive CTE contains at minimum two queries (also known as *members*). The first query that appears in the preceding CTE's body is known as the *Anchor Member.* The anchor member is merely a query that returns a valid table and is used as the basis or anchor for the recursion. In our case, the anchor member simply returns the row for the input root employee (employee 5). The second query that appears in the preceding CTE's body is known as the *Recursive Member.* What makes the query a recursive member is a recursive reference to the CTE's name—EmpsCTE. Note that this reference is not the same as the reference to the CTE's name in the outer query. The reference in the outer query gets the final result table returned by the CTE, and it involves no recursion. However, the inner reference is made before the CTE's result table is finalized, and it is the key element that triggers the recursion. This inner reference to the CTE's name stands for "the previous result set," loosely speaking. In the first invocation of the recursive member, the reference to the CTE's name represents the result set returned from the anchor member. In our case, the recursive member returns subordinates of the employees returned in the previous result set—in other words, the next level of employees.

There's no explicit termination check for the recursion; rather, recursion stops as soon as the recursive member returns an empty set. Because the first invocation of the recursive member yielded a nonempty set (employees 6, 7, and 9), it is invoked again. The second time the recursive member is invoked, the reference to the CTE's name represents the result set returned by the previous invocation of the recursive member (employees 6, 7, and 9). Because these employees have no subordinates, the second invocation of the recursive member yields an empty set, and recursion stops.

The reference to the CTE's name in the outer query stands for the unified (concatenated) results sets of the invocation of the anchor member and all the invocations of the recursive member.

If you run the same code providing employee 2 as input instead of employee 5, you will get the result shown in Table 4-17.

Table 4-17: Subordinates of Employee 2

EmployeeID	ReportsTo	FirstName	LastName
2	NULL	Andrew	Fuller
1	2	Nancy	Davolio
3	2	Janet	Leverling
4	2	Margaret	Peacock
5	2	Steven	Buchanan
8	2	Laura	Callahan
6	5	Michael	Suyama
7	5	Robert	King
9	5	Anne	Dodsworth

Here, the anchor member returns the row for employee 2. The first invocation of the recursive member returns direct subordinates of employee 2: employees 1, 3, 4, 5, and 8. The second invocation of the recursive member returns direct subordinates of employees 1, 3, 4, 5, and 8: employees 6, 7, and 9. The third invocation of the recursive member returns an empty set, and recursion stops. The outer query returns the unified result sets with the rows for employees: 2, 1, 3, 4, 5, 8, 6, 7, and 9.

If you suspect that your data might contain cycles, you can specify the MAXRECURSION hint as a safety measure to limit the number of invocations of the recursive member. You specify the hint right after the outer query:

```
WITH cte_name AS (cte_body) outer_query OPTION(MAXRECURSION n);
```

In this line of code, *n* is the limit for the number of recursive iterations. As soon as the limit is exceeded, the query breaks and an error is generated. Note that MAXRECURSION is set to 100 by default. If you want to remove this limit, specify MAXRECURSION 0. This setting can be specified at the query level only; there's no session, database, or server-level option that you can set to change the default.

To understand how SQL Server processes the recursive CTE, examine the execution plan in Figure 4-4, which was produced for the earlier query returning subordinates of employee 5.

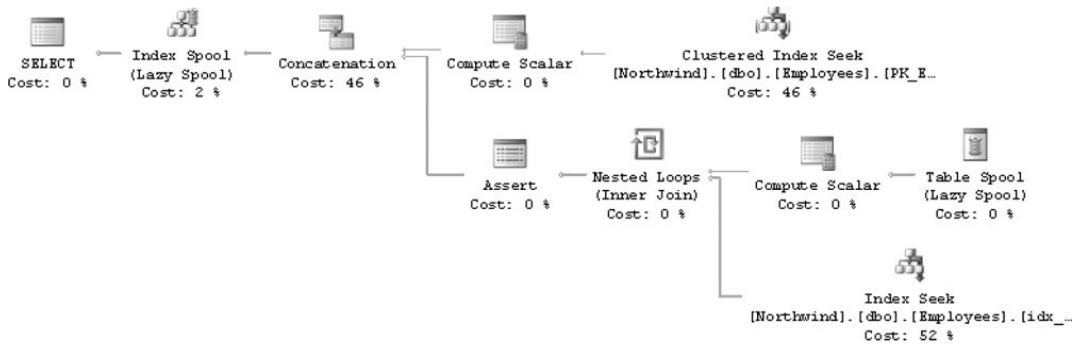

Figure 4-4: Execution plan for recursive CTE

As you can see in the plan, the result set of the anchor member (the row for employee 5) is retrieved using a clustered index seek operation (on the *EmployeeID* column). The *Compute Scalar* operator calculates an iteration counter, which is set to 0 initially (at the first occurrence of *Compute Scalar* in the plan) and incremented by one with each iteration of the recursive member (the second occurrence of *Compute Scalar* in the plan).

Each interim set generated by both the anchor member and the recursive member is spooled in a hidden temporary table (the *Table Spool* operator).

You can also notice later in the plan that a temporary index is created (indicated by the *Index Spool* operator). The index is created on the iteration counter plus the attributes retrieved (*EmployeeID*, *ReportsTo*, *FirstName*, *LastName*).

The interim set of each invocation of the recursive member is retrieved using index seek operations in the covering index I created for the query. The *Nested Loops* operator invokes a seek for each manager returned and spooled in the previous level, to fetch its direct subordinates.

The *Assert* operator checks whether the iteration counter exceeds 100 (the default MAXRECURSION limit). This is the operator in charge of breaking the query in case the number of recursive member invocations exceeds the MAXRECURSION limit.

The *Concatenation* operator concatenates (unifies) all interim result sets.

When you're done testing and experimenting with the recursive CTE, drop the index created for this purpose:

```
DROP INDEX dbo.Employees.idx_mgr_emp_ifname_ilname;
```

Analytical Ranking Functions

SQL Server 2005 introduces four new analytical ranking functions: ROW_NUMBER, RANK, DENSE_RANK, and NTILE. These functions provide a simple and highly efficient way to produce ranking calculations.

ROW_NUMBER is by far my favorite enhancement in SQL Server 2005. Even though it might not seem that significant on the surface compared to other enhancements (for example, recursive queries), it has an amazing number of practical applications that extend far beyond classic ranking and scoring calculations. I have been able to optimize many solutions by using the ROW_NUMBER function, as I will demonstrate throughout the book.

Even though the other ranking functions are technically calculated similarly to ROW_NUMBER underneath the covers, they have far fewer practical applications. RANK and DENSE_RANK are mainly used for ranking and scoring purposes. NTILE is used for more analytical purposes.

The need for such calculations has always existed, but the techniques to achieve them suffered from one or more limitations—they were dramatically slow, complex, or nonstandard. Because this book is intended for both SQL Server 2000 and 2005 users, I'll present the techniques for each version. Because all ranking functions are technically calculated underneath the covers in similar ways, I'll spend most of this section's space on row numbers and follow mainly with the logical aspects of the other functions. After covering the techniques for calculating row numbers, I'll present a benchmark that compares their performance with different table sizes. In my examples, I'll use a Sales table, which you should create and populate by running the code in Listing 4-3.

Listing 4-3: Creating and populating the Sales table

```
SET NOCOUNT ON;
```

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

CREATE TABLE dbo.Sales
(
  empid VARCHAR(10) NOT NULL PRIMARY KEY,
  mgrid VARCHAR(10) NOT NULL,
  qty   INT         NOT NULL
);

INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('A', 'Z', 300);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('B', 'X', 100);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('C', 'X', 200);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('D', 'Y', 200);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('E', 'Z', 250);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('F', 'Z', 300);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('G', 'X', 100);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('H', 'Y', 150);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('I', 'X', 250);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('J', 'Z', 100);
INSERT INTO dbo.Sales(empid, mgrid, qty) VALUES('K', 'Y', 200);

CREATE INDEX idx_qty_empid ON dbo.Sales(qty, empid);
CREATE INDEX idx_mgrid_qty_empid ON dbo.Sales(mgrid, qty, empid);
```

The content of the Sales table returned by the following query is shown in Table 4-18:

Table 4-18: Content of Sales Table

empid	mgrid	qty
A	Z	300
B	X	100
C	X	200
D	Y	200
E	Z	250
F	Z	300
G	X	100
H	Y	150
I	X	250
J	Z	100
K	Y	200

```
SELECT * FROM dbo.Sales;
```

The SQL Server 2005 ranking functions can appear only in the SELECT and ORDER BY clauses of a query. The general form of a ranking function is as follows:

```
ranking_function OVER([PARTITION BY col_list] ORDER BY col_list)
```

The optional PARTITION BY clause allows you to request that the ranking values will be calculated for each partition (or group) of rows separately. For example, if you specify *mgrid* in the PARTITION BY clause, the ranking values will be calculated independently for each manager's rows. In the ORDER BY clause, you specify the column list that determines the order of assignment of the ranking values.

The optimal index for ranking calculations (regardless of the method you use) is one created on *partitioning_columns*, *sort_columns*, *covered_cols.* I created optimal indexes on the Sales table for several ranking calculation requests.

Row Number

Row numbers are sequential integers assigned to rows of a query's result set based on a specified ordering. In the following sections, I'll describe the tools and techniques to calculate row numbers in both SQL Server 2005 and in earlier versions.

The ROW_NUMBER Function in SQL Server 2005

The ROW_NUMBER function assigns sequential integers to rows of a query's result set based on a specified order, optionally within partitions. For example, the following query (which produces the output shown in Table 4-19) returns employee sales rows and assigns row numbers in order of *qty*:

Table 4-19: Row Numbers in Order of *qty*

empid	qty	rownum
B	100	1
G	100	2
J	100	3
H	150	4
K	200	5
C	200	6
D	200	7
E	250	8
I	250	9
F	300	10
A	300	11

```
SELECT empid, qty,
  ROW_NUMBER() OVER(ORDER BY qty) AS rownum
FROM dbo.Sales
ORDER BY qty;
```

To understand the efficiency of the ranking functions in SQL Server 2005, examine the execution plan shown in Figure 4-5, which was generated for this query.

Figure 4-5: Execution plan for ROW_NUMBER

To calculate ranking values, the optimizer needs the data to be sorted first on the partitioning column or columns and then on the sort column or columns.

If you have an index that already maintains the data in the required order, the leaf level of the index is simply scanned in an ordered fashion (as in our case). Otherwise, the data will be scanned and then sorted with a sort operator. The *Sequence Project* operator is the operator in charge of calculating the ranking values. For each input row, it needs two "flags":

1. Is the row the first in the partition? If it is, the *Sequence Project* operator will reset the ranking value.
2. Is the sorting value in this row different than in the previous one? If it is, the *Sequence Project* operator will increment the ranking value as dictated by the specific ranking function.

For all ranking functions, a *Segment* operator will produce the first flag value.

The *Segment* operator basically determines grouping boundaries. It keeps one row in memory and compares it with the next. If they are different, it emits one value. If they are the same, it emits a different value.

To generate the first flag, which indicates whether the row is the first in the partition, the *Segment* operator compares the

PARTITON BY column values of the current and previous rows. Obviously, it emits "true" for the first row read. From the second row on, its output depends on whether the PARTITION BY column value changed. In our example, I didn't specify a PARTITION BY clause, so the whole table is treated as one partition. In this case, *Segment* will emit "true" for the first row and "false" for all others.

As for the second flag (which answers, "Is the value different than the previous value?"), the operator that will calculate it depends on which ranking function you requested. For ROW_NUMBER, the ranking value must be incremented for each row regardless of whether the sort value changes. So in our case, a plain *Compute Scalar* operator simply emits "true" (1) all the time. In other cases (for example, with the RANK and DENSE_RANK functions), another *Segment* operator will be used to tell the *Sequence Project* operator whether the sort value changed in order to determine whether to increment the ranking value or not.

The brilliance of this plan and the techniques the optimizer uses to calculate ranking values might not be apparent yet. For now, it suffices to say that the data is scanned only once, and if it's not already sorted within an index, it is also sorted. This is much faster than any technique that was available to calculate ranking values in SQL Server 2000, as I will demonstrate in detail shortly.

Determinism As you probably noticed in the output of the previous query, row numbers keep incrementing regardless of whether the sort value changes or not. Row numbers must be unique within the partition. This means that for a nonunique sort list, the query is nondeterministic. That is, there are different result sets that are correct and not just one. For example, in Table 4-19 you can see that employees B, G, and J, all having a quantity of 100, got the row numbers 1, 2, and 3, respectively. However, the result would also be valid if these three employees received the row numbers 1, 2, and 3 in a different order.

For some applications determinism is mandatory. To guarantee determinism, you simply need to add a tiebreaker that makes the values of *partitioning column(s) + sort column(s)* unique.

For example, the following query (which generates the output shown in Table 4-20) demonstrates both a nondeterministic row number based on the *qty* column alone and also a deterministic one based on the order of *qty* and *empid*:

Table 4-20: Row Numbers, Determinism

empid	qty	nd_rownum	d_rownum
B	100	1	1
G	100	2	2
J	100	3	3
H	150	4	4
C	200	6	5
D	200	7	6
K	200	5	7
E	250	8	8
I	250	9	9
A	300	11	10
F	300	10	11

```
SELECT empid, qty,
  ROW_NUMBER() OVER(ORDER BY qty)        AS nd_rownum,
  ROW_NUMBER() OVER(ORDER BY qty, empid) AS d_rownum
FROM dbo.Sales
ORDER BY qty, empid;
```

Partitioning As I mentioned earlier, you can also calculate ranking values within partitions (groups of rows). The following example (which generates the output shown in Table 4-21) calculates row numbers based on the order of *qty* and *empid*, for each manager separately:

Table 4-21: Row Numbers, Partitioned

mgrid	empid	qty	rownum

X	B	100	1
X	G	100	2
X	C	200	3
X	I	250	4
Y	H	150	1
Y	D	200	2
Y	K	200	3
Z	J	100	1
Z	E	250	2
Z	A	300	3
Z	F	300	4

```
SELECT mgrid, empid, qty,
  ROW_NUMBER() OVER(PARTITION BY mgrid ORDER BY qty, empid) AS rownum
FROM dbo.Sales
ORDER BY mgrid, qty, empid;
```

The Set-Based Technique prior to SQL Server 2005

There are several techniques for calculating ranking values using a version of SQL Server prior to SQL Server 2005, and all of them suffer from some limitation. Before I start describing these techniques, keep in mind that you can also calculate ranking values at the client. Whatever way you choose, your client will iterate through the records in the recordset returned from SQL Server. The client can simply request the rows sorted and, in a loop, increment a counter. Of course, if you need the ranking values for further server-side manipulation before results are sent to the client, client-side ranking is not an option.

I'll start with the standard and set-based technique. Unfortunately, it is usually the slowest of all.

Unique Sort Column Prior to SQL Server 2005, reasonably simple set-based calculations of row numbers were possible, given a unique *partitioning + sort column(s)* combination. As I will describe later, set-based row number calculations without this unique combination also exist, but they are substantially more complex.

All ranking value calculations can be achieved by counting rows. To calculate row numbers, you can employ the following fundamental technique. You simply use a subquery to count the number of rows with a smaller or equal sort value. This count corresponds to the desired row number. For example, the following query produces row numbers based on *empid*, generating the output in Table 4-22:

Table 4-22: Row Numbers, Unique Sort Column

empid	rownum
A	1
B	2
C	3
D	4
E	5
F	6
G	7
H	8
I	9
J	10
K	11

```
SELECT empid,
```

```
  (SELECT COUNT(*)
   FROM dbo.Sales AS S2
   WHERE S2.empid <= S1.empid) AS rownum
FROM dbo.Sales AS S1
ORDER BY empid;
```

This technique to calculate row numbers, though fairly simple, is extremely slow. To understand why, examine the execution plan shown in Figure 4-6 created for the query.

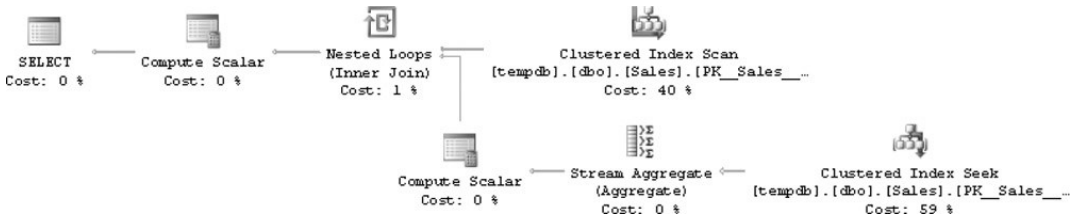

Figure 4-6: Execution plan for pre–SQL Server 2005, set-based, row number query

There is an index on the sort column (*empid*) that happens to be the Sales table's clustered index. The table is first fully scanned (*Clustered Index Scan* operator) to return all rows.

For each row returned from the initial full scan, the *Nested Loops* operator invokes the activity that generates the row number by counting rows. Each row number calculation involves a seek operation within the clustered index, followed by a partial scan operation (from the head of the leaf level's linked list to the last point where *inner_empid* is smaller than or equal to *outer_empid*).

Note that there are two different operators that use the clustered index—first, a full scan to return all rows; second, a seek followed by a partial scan for each outer row to achieve the count.

Remember that the primary factor affecting the performance of queries that do data manipulation will usually be I/O. A rough estimate of the number or rows accessed here will show how inefficient this execution plan is. To calculate *rownum* for the first row of the table, SQL Server needs to scan 1 row in the index. For the second row, it needs to scan 2 rows. For the third row, it needs to scan 3 rows, and so on, and for the n^{th} row of the table, it needs to scan *n* rows. For a table with *n* rows, having an index based on the sort column in place, the total number of rows scanned is 1 + 2 + 3 + … + *n.* You may not grasp immediately the large number of rows that are going to be scanned. To give you a sense, for a table with 100,000 rows, you're looking at 5,000,050,000 rows that are going to be scanned in total.

As an aside, there's a story told about the mathematician Gauss. When he was a child, he and his classmates got an assignment from their teacher to find the sum of all the integers from 1 through 100. Gauss gave the answer almost instantly. When the teacher asked him how he came up with the answer so fast, he said that he added the first and the last values (1+100 =101), and then multiplied that total by half the number of integers (50), which is the number of pairs. Sure enough, the result of *first_val* + *last_val* is equal to the *second_val* + *next_to_last val*, and so on. In short, the formula for the sum of the first n positive integers is $(n + n^2)/2$. That's the number of rows that need to be scanned in total to calculate row numbers using this technique when there is an index based on the sort column. You're looking at an n^2 graph of I/O cost and run time based on the number of rows in the table. You can play with the numbers in the formula and see that the cost gets humongous pretty quickly.

When there's no index on the table, matters are even worse. To calculate each row number, the entire table needs to be scanned. The total number of rows scanned by the query is then n^2. For example, given a table with 100,000 rows, the query will end up scanning 10,000,000,000 rows in total.

Nonunique Sort Column and Tiebreaker When the sort column is not unique, you can make it unique by introducing a tiebreaker, to allow a solution that keeps a reasonable level of simplicity. Let *sortcol* be the sort column, and let *tiebreaker* be the tiebreaker column. To count rows with the same or smaller values of the sort list (*sortcol+tiebreaker*), use the following expression in the subquery:

```
inner_sortcol < outer_sortcol
OR inner_sortcol = outer_sortcol
   AND inner_tiebreaker < = outer_tiebreaker
```

Note that operator precedence dictates that AND would be evaluated prior to OR. For clarity, manageability, and

readability you might want to use parentheses.

The following query (which generates the output shown in Table 4-23) produces row numbers based on *qty* and *empid*, in that order:

Table 4-23: Row Numbers, Nonunique Sort Column, and Tiebreaker

empid	qty	rownum
B	100	1
G	100	2
J	100	3
H	150	4
C	200	5
D	200	6
K	200	7
E	250	8
I	250	9
A	300	10
F	300	11

```
SELECT empid, qty,
  (SELECT COUNT(*)
   FROM dbo.Sales AS S2
   WHERE S2.qty < S1.qty
      OR (S2.qty = S1.qty AND S2.empid < = S1.empid)) AS row num
FROM dbo.Sales AS S1
ORDER BY qty, empid;
```

Nonunique Sort Column without a Tiebreaker The problem becomes substantially more complex when you need to assign row numbers according to a nonunique sort column and using no tiebreaker, with a pre–SQL Server 2005 set-based technique. For example, given the table T1, which you create and populate by running the code in Listing 4-4, say you are supposed to produce row numbers based on *col1* ordering.

Listing 4-4: Creating and populating the T1 table

```
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1;
GO
CREATE TABLE dbo.T1(col1 VARCHAR(5));
INSERT INTO dbo.T1(col1) VALUES('A');
INSERT INTO dbo.T1(col1) VALUES('A');
INSERT INTO dbo.T1(col1) VALUES('A');
INSERT INTO dbo.T1(col1) VALUES('B');
INSERT INTO dbo.T1(col1) VALUES('B');
INSERT INTO dbo.T1(col1) VALUES('C');

INSERT INTO dbo.T1(col1) VALUES('C');
INSERT INTO dbo.T1(col1) VALUES('C');
INSERT INTO dbo.T1(col1) VALUES('C');
INSERT INTO dbo.T1(col1) VALUES('C');
```

The solution must be compatible with SQL Server 2000, so you can't simply use the ROW_NUMBER function. Also, the solution must be standard.

In the solution for this problem, I'll make first use of a very important key technique—generating duplicate rows using an auxiliary table of numbers. More accurately, the technique generates sequentially numbered copies of each row, but for simplicity's sake, I'll refer to it throughout the book as "generating duplicates."

I'll explain the concept of the auxiliary table of numbers and how to create one later in the chapter in the section "Auxiliary Table of Numbers." For now, simply run the code in Listing 4-8, which creates the Nums table and populates it with the 1,000,000 integers in the range *1* <= *n* <= *1,000,000.*

The technique that I'm talking about is "generating duplicates" or "expanding" the number of rows. For example, given a table T1, say you want to generate 5 copies of each row. To achieve this, you can use the Nums table as follows:

```
SELECT ... FROM dbo.T1, dbo.Nums WHERE n <= 5;
```

I will provide more details on the "generating duplicates" technique and its uses in Chapter 5.

Going back to our original problem, you're supposed to generate row numbers for the rows of T1, based on *col1* order. The first step in the solution is "collapsing" the rows by grouping them by *col1.* For each group, you return the number of duplicates (a count of rows in the group). You also return, using a subquery, the number of rows in the base table that have a smaller sort value. Here's the query that accomplishes the first step, and its output is shown in Table 4-24:

Table 4-24: Row Numbers, Nonunique Sort Column, No Tiebreaker, Step 1's Output

col1	dups	smaller
A	3	0
B	2	3
C	5	5

```
SELECT col1, COUNT(*) AS dups,
  (SELECT COUNT(*) FROM dbo.T1 AS B
   WHEREB.col1 < A.col1) AS smaller
FROM dbo.T1 AS A
GROUP BY col1;
```

For example, A appears 3 times, and there are 0 rows with a *col1* value smaller than A. B appears 2 times, and there are 3 rows with a *col1* value smaller than B. And so on.

The next step (which produces the output shown in Table 4-25) is to expand the number of rows or create sequentially numbered copies of each row. You achieve this by creating a derived table out of the previous query and joining it to the Nums table as follows, based on *n* <= *dups*:

Table 4-25: Row Numbers, Nonunique Sort Column, No Tiebreaker, Step 2's Output

col1	dups	smaller	n
A	3	0	1
A	3	0	2
A	3	0	3
B	2	3	1
B	2	3	2
C	5	5	1
C	5	5	2
C	5	5	3
C	5	5	4
C	5	5	5

```
SELECT col1, dups, smaller, n
FROM (SELECT col1, COUNT(*) AS dups,
```

```
        (SELECT COUNT(*) FROM dbo.T1 AS B
         WHERE B.col1 < A.col1) AS smaller
      FROM dbo.T1 AS A
      GROUP BY col1) AS D, Nums
WHERE n <= dups;
```

Now look carefully at the output in Table 4-25, and see whether you can figure out how to produce the row numbers.

The row number can be expressed as the number of rows with a smaller sort value, plus the row number within the same sort value group—in other words, *n + smaller.* The following query is the final solution, and it generates the output shown in Table 4-26:

Table 4-26: Row Numbers, Nonunique Sort Column, No Tiebreaker, Final Output

rownum	col1
1	A
2	A
3	A
4	B
5	B
6	C
7	C
8	C
9	C
10	C

```
SELECT n + smaller AS rownum, col1
FROM (SELECT col1, COUNT(*) AS dups,
        (SELECT COUNT(*) FROM dbo.T1 AS B
         WHERE B.col1 < A.col1) AS smaller
      FROM dbo.T1 AS A
      GROUP BY col1) AS D, Nums
WHERE n <= dups;
```

Partitioning Partitioning is achieved by simply adding a correlation in the subquery based on a match between the partitioning column or columns in the inner and outer tables. For example, the following query against the Orders table (which generates the output shown in Table 4-27) calculates row numbers that are partitioned by *mgrid*, ordered by *qty*, and use *empid* as a tiebreaker:

Table 4-27: Row Numbers, Partitioned

mgrid	empid	qty	rownum
X	B	100	1
X	G	100	2
X	C	200	3
X	I	250	4
Y	H	150	1
Y	D	200	2
Y	K	200	3
Z	J	100	1
Z	E	250	2

Z	A	300	3
Z	F	300	4

```
SELECT mgrid, empid, qty,
  (SELECT COUNT(*)
   FROM dbo.Sales AS S2
   WHERE S2.mgrid = S1.mgrid
     AND (S2.qty < S1.qty
          OR (S2.qty = S1.qty AND S2.empid < = S1.empid))) AS rownum
FROM dbo.Sales AS S1
ORDER BY mgrid, qty, empid;
```

Note As I mentioned earlier, the pre–SQL Server 2005 set-based technique to calculate row numbers has a num_rows^2 cost. However, for a fairly small number of rows (in the area of dozens), it's pretty fast. Note that the performance problem has more to do with the partition size rather than the table's size. If you create the recommended index based on *partitioning_cols, sort_cols, tiebreaker_cols*, the number of rows scanned within the index is equivalent to the row number generated. The row number is reset (starts from 1) with every new partition. So even for very large tables, when the partition size is fairly small and you have a proper index in place, the solution is pretty fast.

If you have *p* partitions and *r* rows in each partition, the number of rows scanned in total is: $p * (r + r^2)/2$. For example, if you have 100,000 partitions, and 10 rows in each partition, you get 5,500,000 rows scanned in total. Though this number might seem large, it's nowhere near the number you get without partitioning. And as long as the partition size remains constant, the graph of query cost compared with the number of rows in the table is linear.

Cursor-Based Solution

You can use a cursor to calculate row numbers. A cursor-based solution for any of the aforementioned variations is pretty straightforward. You create a fast-forward (read-only, forward-only) cursor based on a query that orders the data by *partitioning_cols*, *sort_cols*, *tiebreaker_cols.* As you fetch rows from the cursor, you simply increment a counter, resetting it every time a new partition is detected. You can store the result rows along with the row numbers in a temporary table or a table variable.

As an example, the code in Listing 4-5 (which generates the output shown in Table 4-28) uses a cursor to calculate row numbers based on the order of *qty* and *empid*:

Table 4-28: Output of Cursor Calculating Row Numbers

empid	qty	rn
B	100	1
G	100	2
J	100	3
H	150	4
C	200	5
D	200	6
K	200	7
E	250	8
I	250	9
A	300	10
F	300	11

Listing 4-5: Calculating row numbers with a cursor

```
DECLARE @SalesRN TABLE(empid VARCHAR(5), qtyINT, rnINT);
DECLARE @empid AS VARCHAR(5), @qty AS INT, @rn AS INT;
```

```
BEGIN TRAN

DECLARE rncursor CURSOR FAST_FORWARD FOR
  SELECT empid, qty FROM dbo.Sales ORDER BY qty, empid;
OPEN rncursor;

SET @rn = 0;

FETCH NEXT FROM rncursor INTO @empid, @qty;
WHILE @@fetch_status = 0
BEGIN
  SET @rn = @rn + 1;
  INSERT INTO @SalesRN(empid, qty, rn) VALUES(@empid, @qty, @rn);
  FETCH NEXT FROM rncursor INTO @empid, @qty;
END

CLOSE rncursor;
DEALLOCATE rncursor;

COMMIT TRAN

SELECT empid, qty, rn FROM @SalesRN;
```

Generally, working with cursors should be avoided, as cursors have a lot of overhead that is a drag on performance. However, in this case, unless the partition size is really tiny, the cursorbased solution performs much better than the pre–SQL Server 2005 set-based technique, as it scans the data only once. This means that as the table grows larger, the cursor-based solution has a linear performance degradation, as opposed to the n^2 one that the pre–SQL Server 2005 set-based solution has.

IDENTITY-Based Solution

In versions prior to SQL Server 2005, you can also rely on the IDENTITY function and IDENTITY column property to calculate row numbers. As such, calculating row numbers with IDENTITY is a useful technique to know. Before you proceed, though, you should be aware that when you use the IDENTITY function, you cannot guarantee the order of assignment of IDENTITY values. You can, however, guarantee the order of assignment by using an IDENTITY column instead of the IDENTITY function: first create a table with an IDENTITY column, and then load the data using an INSERT SELECT statement with an ORDER BY clause.

More Info You can find a detailed discussion of IDENTITY and ORDER BY in Knowledge Base article 273586 (http://www.support.microsoft.com/default.aspx?scid=kb; en-us;273586), which I strongly recommend that you read.

Nonpartitioned Using the IDENTITY function in a SELECT INTO statement is by far the fastest way to calculate row numbers at the server prior to SQL Server 2005. The first reason for this is that you scan the data only once, without the overhead involved with cursor manipulation. The second reason is that SELECT INTO is a minimally logged operation when the database recovery model is not FULL. However, keep in mind that you can trust it only when you don't care about the order of assignment of the row numbers.

For example, the following code demonstrates how to use the IDENTITY function to create and populate a temporary table with row numbers, in no particular order:

```
SELECT empid, qty, IDENTITY(int, 1, 1) AS rn
INTO #SalesRN FROM dbo.Sales;

SELECT * FROM #SalesRN;

DROP TABLE #SalesRN;
```

This technique is handy when you need to generate integer identifiers to distinguish rows for some processing need.

Don't let the fact that you can technically specify an ORDER BY clause in the SELECT INTO query mislead you. In SQL Server 2000, there's no guarantee that in the execution plan the assignment of IDENTITY values will take place after the sort. In SQL Server 2005, it's irrelevant because you will use the ROW_NUMBER function anyway.

As mentioned earlier, when you do care about the order of assignment of the IDENTITY values—in other words, when the

row numbers should be based on a given order—first create the table, and then load the data. This technique is not as fast as the SELECT INTO approach because INSERT SELECT is always fully logged; however, it's still much faster than the other techniques available prior to SQL Server 2005.

Here's an example for calculating row numbers based on the order of *qty* and *empid*:

```
CREATE TABLE #SalesRN(empid VARCHAR(5), qty INT, rn INT IDENTITY);

INSERT INTO #SalesRN(empid, qty)
  SELECT empid, qty FROM dbo.Sales ORDER BY qty, empid;

SELECT * FROM #SalesRN;

DROP TABLE #SalesRN;
```

Partitioned Using the IDENTITY approach to create partitioned row numbers requires an additional step. As with the nonpartitioned solution, you insert the data into a table with an IDENTITY column, only this time it is sorted by *partitioning_cols*, *sort_cols*, *tiebreaker_cols.*

The additional step is a query that calculates the row number within the partition using the following formula: *general_row_number—min_row_number_within_partition + 1.* The minimum row number within the partition can be obtained by either a correlated subquery or a join.

As an example, the code in Listing 4-6 generates row numbers partitioned by *mgrid*, sorted by *qty* and *empid.* The code presents both the subquery approach and the join approach to obtaining the minimum row number within the partition.

Listing 4-6: Calculating partitioned row numbers with an IDENTITY

```
CREATE TABLE #SalesRN
  (mgrid VARCHAR(5), empid VARCHAR(5), qty INT, rn INT IDENTITY);
CREATE UNIQUE CLUSTERED INDEX idx_mgrid_rn ON #SalesRN(mgrid, rn);

INSERT INTO #SalesRN(mgrid, empid, qty)
  SELECT mgrid, empid, qty FROM dbo.Sales ORDER BY mgrid, qty, empid;

--Option 1 - usinga subquery
SELECT mgrid, empid, qty,
  rn - (SELECT MIN(rn) FROM #SalesRN AS S2
        WHERE S2.mgrid = S1.mgrid) + 1 AS rn
FROM #SalesRN AS S1;

-- Option 2 - usinga join
SELECT S.mgrid, empid, qty, rn - minrn + 1 AS rn
FROM #SalesRN AS S
  JOIN (SELECT mgrid, MIN(rn) AS minrn
        FROM #SalesRN
        GROUP BY mgrid) AS M
    ONS.mgrid = M.mgrid;

DROP TABLE #SalesRN;
```

Performance Comparisons

I presented four different techniques to calculate row numbers server-side. One is available only in SQL Server 2005 (using the ROW_NUMBER function), and three (set-based using subqueries, cursor-based, and IDENTITY-based) are available in both versions. As I mentioned earlier, the other three ranking calculations that I'll describe later in this chapter are technically calculated using very similar access methods. So the performance aspects that I discussed and the following benchmark that I'll present are relevant to all ranking calculations.

I ran the benchmark shown in Listing 4-7 on my laptop (single CPU: Intel Centrino 1.7 Mhz; RAM: 1GB, single disk drive). Even though my laptop is not exactly the best model for a production server, you can get a good sense of the performance differences between the techniques. The benchmark populates a table with increasing numbers of rows, starting with 10,000 and progressing up to 100,000 in steps of 10,000 rows. The benchmark calculates row numbers using all four techniques, with the Discard Results option turned on in SQL Server Management Studio (SSMS) to remove the effect of generating the output. The benchmark records the run times in milliseconds in the RNBenchmark table.

Listing 4-7: Benchmark comparing techniques to calculate row numbers

```
-- Change Tool's Options to Discard Query Results
SET NOCOUNT ON;
USE tempdb;
GO
IF OBJECT_ID('dbo.RNBenchmark') IS NOT NULL
  DROP TABLE dbo.RNBenchmark;
GO
IF OBJECT_ID('dbo.RNTechniques') IS NOT NULL
  DROP TABLE dbo.RNTechniques;
GO
IF OBJECT_ID('dbo.SalesBM') IS NOT NULL
  DROP TABLE dbo.SalesBM;
GO
IF OBJECT_ID('dbo.SalesBMIdentity') IS NOT NULL
  DROP TABLE dbo.SalesBMIdentity;
GO
IF OBJECT_ID('dbo.SalesBMCursor') IS NOT NULL
  DROP TABLE dbo.SalesBMCursor;
GO

CREATE TABLE dbo.RNTechniques
(
  tid INT NOT NULL PRIMARY KEY,
  technique VARCHAR(25) NOT NULL
);
INSERT INTO RNTechniques(tid, technique) VALUES(1, 'Set-Based 2000');
INSERT INTO RNTechniques(tid, technique) VALUES(2, 'IDENTITY');
INSERT INTO RNTechniques(tid, technique) VALUES(3, 'Cursor');
INSERT INTO RNTechniques(tid, technique) VALUES(4, 'ROW_NUMBER 2005');
GO

CREATE TABLE dbo.RNBenchmark
(
  tid       INT    NOT NULL REFERENCES dbo.RNTechniques(tid),
  numrows   INT    NOT NULL,
  runtimems BIGINT NOT NULL,
  PRIMARY KEY(tid, numrows)
);
GO

CREATE TABLE dbo.SalesBM
(
  empid INT NOT NULL IDENTITY PRIMARY KEY,
  qty   INT NOT NULL
);
CREATE INDEX idx_qty_empid ON dbo.SalesBM(qty, empid);
GO
CREATE TABLE dbo.SalesBMIdentity(empid INT, qty INT, rn INT IDENTITY);
GO
CREATE TABLE dbo.SalesBMCursor(empid INT, qty INT, rn INT);
GO

DECLARE
  @maxnumrows    AS INT,
  @steprows      AS INT,
  @curnumrows    AS INT,
  @dt            AS DATETIME;

SET @maxnumrows    = 100000;
SET @steprows      = 10000;
SET @curnumrows    = 10000;

WHILE @curnumrows <= @maxnumrows
BEGIN
```

```
TRUNCATE TABLE dbo.SalesBM;
INSERT INTO dbo.SalesBM(qty)
  SELECT CAST(1+999.9999999999*RAND(CHECKSUM(NEWID())) AS INT)
  FROM dbo.Nums
  WHERE n <= @curnumrows;

-- 'Set-Based 2000'

DBCC FREEPROCCACHE WITH NO_INFOMSGS;
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

SET @dt = GETDATE();

SELECT empid, qty,
  (SELECT COUNT(*)
   FROM dbo.SalesBM AS S2
   WHERE S2.qty < S1.qty
       OR (S2.qty = S1.qty AND S2.empid <= S1.empid)) AS rn
FROM dbo.SalesBM AS S1
ORDER BY qty, empid;

INSERT INTO dbo.RNBenchmark(tid, numrows, runtimems)
  VALUES(1, @curnumrows, DATEDIFF(ms, @dt, GETDATE()));

-- 'IDENTITY'

TRUNCATE TABLE dbo.SalesBMIdentity;

DBCC FREEPROCCACHE WITH NO_INFOMSGS;
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

SET @dt = GETDATE();

INSERT INTO dbo.SalesBMIdentity(empid, qty)
  SELECT empid, qty FROM dbo.SalesBM ORDER BY qty, empid;

SELECT empid, qty, rn FROM dbo.SalesBMIdentity;

INSERT INTO dbo.RNBenchmark(tid, numrows, runtimems)
  VALUES(2, @curnumrows, DATEDIFF(ms, @dt, GETDATE()));

-- 'Cursor'

TRUNCATE TABLE dbo.SalesBMCursor;

DBCC FREEPROCCACHE WITH NO_INFOMSGS;
DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

SET @dt = GETDATE();

DECLARE @empid AS INT, @qty AS INT, @rn AS INT;

BEGIN TRAN

DECLARE rncursor CURSOR FAST_FORWARD FOR
  SELECT empid, qty FROM dbo.SalesBM ORDER BY qty, empid;
OPEN rncursor;

SET @rn = 0;

FETCH NEXT FROM rncursor INTO @empid, @qty;
WHILE @@fetch_status = 0
BEGIN
  SET @rn = @rn + 1;
  INSERT INTO dbo.SalesBMCursor(empid, qty, rn)
    VALUES(@empid, @qty, @rn);
  FETCH NEXT FROM rncursor INTO @empid, @qty;
END
```

```
  CLOSE rncursor;
  DEALLOCATE rncursor;

  COMMIT TRAN

  SELECT empid, qty, rn FROM dbo.SalesBMCursor;

  INSERT INTO dbo.RNBenchmark(tid, numrows, runtimems)
    VALUES(3, @curnumrows, DATEDIFF(ms, @dt, GETDATE()));

  -- 'ROW_NUMBER 2005'

  DBCC FREEPROCCACHE WITH NO_INFOMSGS;
  DBCC DROPCLEANBUFFERS WITH NO_INFOMSGS;

  SET @dt = GETDATE();

  SELECT empid, qty, ROW_NUMBER() OVER(ORDERBY qty, empid) AS rn
  FROM dbo.SalesBM;

  INSERT INTO dbo.RNBenchmark(tid, numrows, runtimems)
    VALUES(4, @curnumrows, DATEDIFF(ms, @dt, GETDATE()));

  SET @curnumrows = @curnumrows + @steprows;

END
```

The following query returns the benchmark's results in a conveniently readable format, which is shown in Table 4-29:

Table 4-29: Benchmark Results

numrows	Set-Based 2000	IDENTITY	Cursor	ROW_NUMBER 2005
10000	11960	80	550	30
20000	33260	160	1343	40
30000	72646	660	1613	30
40000	127033	893	2110	60
50000	199740	870	2873	70
60000	283616	990	3043	63
70000	382130	900	3913	103
80000	499580	1123	4276	80
90000	634653	1040	4766	100
100000	796806	1060	5280	140

```
SELECT numrows,
  [Set-Based 2000], [IDENTITY], [Cursor], [ROW_NUMBER 2005]
FROM (SELECT technique, numrows, runtimems
      FROM dbo.RNBenchmark AS B
        JOIN dbo.RNTechniques AS T
          ON B.tid = T.tid) AS D
PIVOT(MAX(runtimems) FOR technique IN(
  [Set-Based 2000], [IDENTITY], [Cursor], [ROW_NUMBER 2005])) AS P
ORDER BY numrows;
```

The query uses a pivoting technique that I'll describe in Chapter 6, so don't try to squeeze your brains if you're not familiar with it. For our discussion, the important thing is the benchmark's results. You can immediately see that the pre–SQL Server 2005 set-based technique is dramatically slower than all the rest, and I explained why earlier. You will also notice that the ROW_NUMBER function is dramatically faster than all the rest. I wanted to present a graph with all results, but the run times of the pre–SQL Server 2005 set-based techniques were so great that the lines for the other solutions were simply flat. So I decided to present two separate graphs. Figure 4-7 shows the graph of run times for the IDENTITY-based, cursor-based, and ROW_NUMBER function-based techniques. Figure 4-8 shows the graph for the set-based pre–SQL Server 2005 technique.

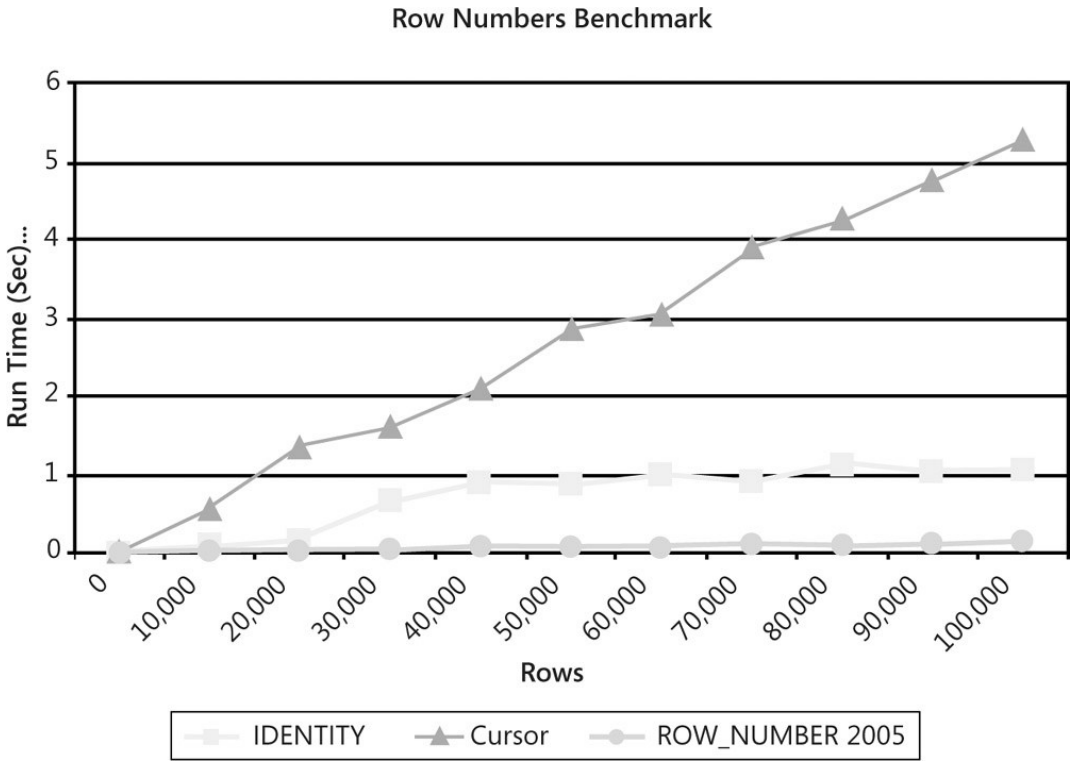

Figure 4-7: Row Numbers benchmark graph I

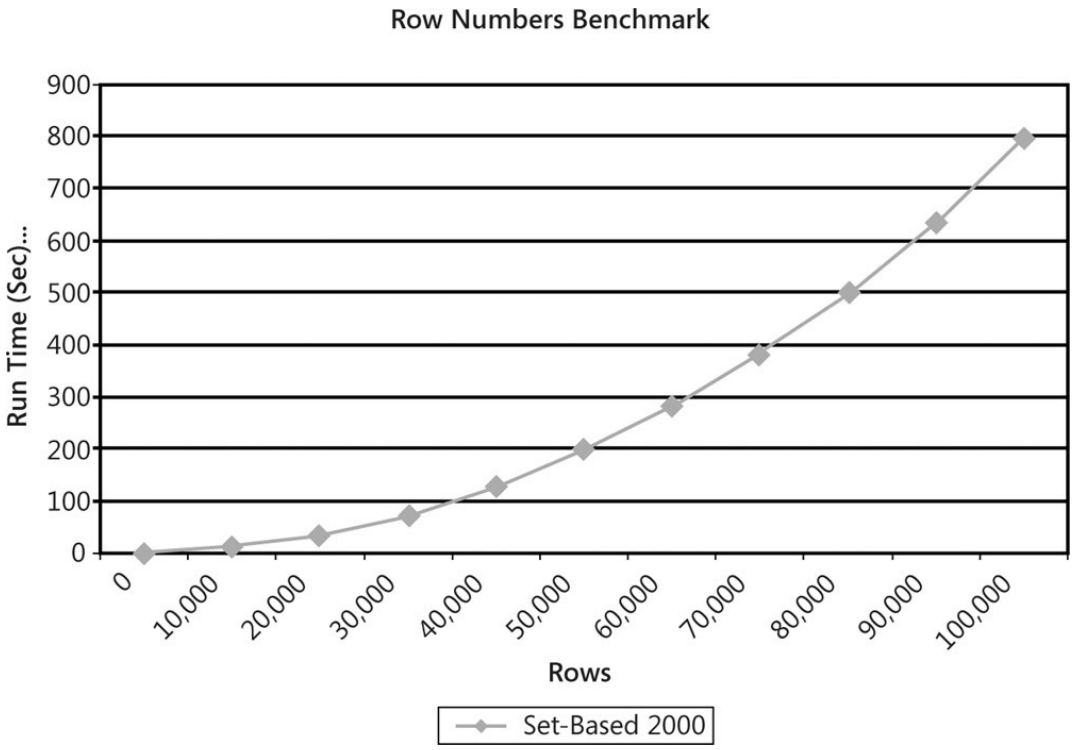

Figure 4-8: Row Numbers benchmark graph II

You can see in Figure 4-7 that all three techniques have a fairly linear performance graph, while Figure 4-8 shows a beautifully curved n^2 graph.

The obvious conclusion is that in SQL Server 2005 you should always use the new ranking functions. In SQL Server 2000, if it's important to you to use a standard set-based technique, only use that technique when the partition size is fairly small (measured in dozens). Otherwise, use the IDENTITY-based technique, first creating the table, and then loading the data.

Paging

As I mentioned earlier, row numbers have many practical applications that I'll demonstrate throughout the book. Here I'd like to show one example where I use row numbers to achieve paging—accessing rows of a result set in chunks. Paging is a common need in applications, allowing the user to navigate through chunks or portions of a result set. Paging with row numbers is also a handy technique. This example will also allow me to demonstrate additional optimization techniques that the optimizer applies when using the ROW_NUMBER function. Of course, in SQL Server 2000 you can use the less efficient techniques to calculate row numbers to achieve paging.

Ad Hoc Paging Ad hoc paging is a request for a single page, where the input is the page number and page size (the number of rows in a page). When the user needs a particular single page and won't request additional pages, you implement a different solution than the one you would for multiple page requests. First you have to realize that there's no way to access page *n* without physically accessing pages 1 through *n*–1. Bearing this in mind, the following code returns a page of rows from the Sales table ordered by *qty* and *empid*, given the page size and page number as inputs:

```
DECLARE @pagesize AS INT, @pagenum AS INT;
SET @pagesize = 5;
SET @pagenum = 2;

WITH SalesCTE AS
(
  SELECT ROW_NUMBER() OVER(ORDER BY qty, empid) AS rownum,
    empid, mgrid, qty
  FROM dbo.Sales
)
SELECT rownum, empid, mgrid, qty
FROM SalesCTE
WHERE rownum > @pagesize * (@pagenum-1)
  AND rownum <= @pagesize * @pagenum
ORDER BY rownum;
```

This code generates the output shown in Table 4-30.

Table 4-30: Second Page of Sales Ordered by *qty*, *empid* with a Page Size of 5 Rows

rownum	empid	mgrid	qty
6	D	Y	200
7	K	Y	200
8	E	Z	250
9	I	X	250
10	A	Z	300

The CTE called SalesCTE assigns row numbers to the sales rows based on the order of *qty* and *empid.* The outer query filters only the target page's rows using a formula based on the input page size and page number.

You might be concerned that the query appears to calculate row numbers for all rows and then filter only the requested page's rows. This might seem to require a full table scan. With very large tables this, of course, would be a serious performance issue. However, before getting concerned, examine the execution plan for this query, which is shown in Figure 4-9.

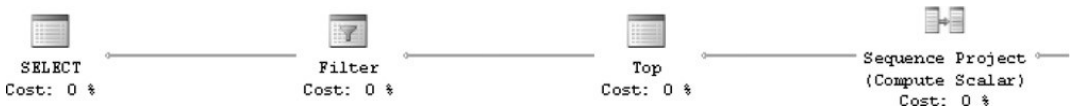

Figure 4-9: Execution plan for the ad hoc paging solution

The figure shows only the left part of the plan starting with the Sequence Project, which assigns the row numbers. If you look at the properties of the *Top* operator, you will see that the plan scans only the first 10 rows of the table. Because the code requests the second page of five rows, only the first two pages are scanned. Then the *Filter* operator filters only the second page (rows 6 through 10).

Another way to demonstrate that the whole table is not scanned is by populating the table with a large number of rows and running the query with the SET STATISTICS IO option turned on. You will notice by the number of reads reported that when you request page *n*, regardless of the size of the table, only the first *n* pages of rows are scanned.

This solution can perform well even when there are multiple page requests that usually "move forward"—that is, page 1 is requested, then page 2, then page 3, and so on. When the first page of rows is requested, the relevant data/index pages are physically scanned and loaded into cache (if they're not there already). When the second page of rows is requested, the data pages for the first request already reside in cache, and only the data pages for the second page of rows need to be physically scanned. This requires mostly logical reads (reads from cache), and physical reads are only needed for the requested page. Logical reads are much faster than physical reads, but keep in mind that they also have a cost that accumulates.

Multipage Access There's another solution for paging that will typically perform better overall than the previous solution when there are multiple page requests that do not move forward, if the result set is not very large. First materialize all pages in a table along with row numbers, and create a clustered index on the row number column:

```
SELECT ROW_NUMBER() OVER(ORDER BY qty, empid) AS rownum,
  empid, mgrid, qty
INTO #SalesRN
FROM dbo.Sales;

CREATE UNIQUE CLUSTERED INDEX idx_rn ON #SalesRN(rownum);
```

Now you can satisfy any page request with a query like the following:

```
DECLARE @pagesize AS INT, @pagenum AS INT;
SET @pagesize = 5;
SET @pagenum = 2;

SELECT rownum, empid, mgrid, qty
FROM #SalesRN
WHERE rownum BETWEEN @pagesize * (@pagenum-1) + 1
                 AND @pagesize * @pagenum
ORDER BY rownum;
```

The execution plan for this query is shown in Figure 4-10 (abbreviated by removing the operators that calculate boundaries up to the *Merge Interval* operator, to focus on the actual data access).

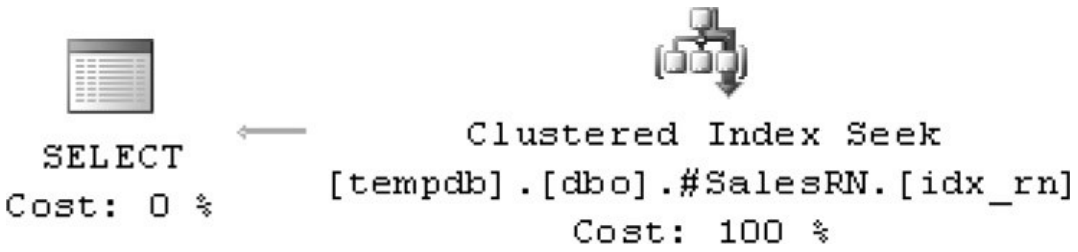

Figure 4-10: Execution plan for multipaging solution

This is a very efficient plan that performs a seek within the index to reach the low boundary row (row number 6 in this case), followed by a partial scan (not visible in the plan), until it reaches the high boundary row (row number 10). Only the rows of the requested page of results are scanned within the index.

If your application design is such that it disconnects after each request, obviously the temporary table will be gone as soon as the creating session disconnects. In such a case, you might want to create a permanent table that is logically "temporary." You can achieve this by naming the table *some_name*<*some_identifier*>—for example, T<guid> (Global Unique Identifier).

You will also need to develop a garbage-collection (cleanup) process that gets rid of tables that the application didn't have a chance to drop explicitly in cases where it terminated in a disorderly way.

In cases where you need to support large result sets or a high level of concurrency, you will have scalability issues related to tempdb resources. You can develop a partitioned solution that materializes only a certain number of pages and not all of them—for example, 1000 rows at a time. Typically, users don't request more than the first few pages anyway. If a user ends up requesting pages beyond the first batch, you can materialize the next partition (that is, the next 1000 rows).

When you don't care about materializing the result set in a temporary table for multipage access, you might want to consider using a table variable where you materialize only the first batch of pages (for example, 1000 rows). Table variables don't involve recompilations, and they suffer less from logging and locking issues. The optimizer doesn't collect statistics for table variables, so you should be very cautious and selective in choosing the cases to use them for. But when all you need to do is store a small result set and scan it entirely anyway, that's fine.

Once you're done using this table, you can drop it:

```
DROP TABLE #SalesRN;
```

Rank and Dense Rank

Rank and dense rank are calculations similar to row number. But unlike row number, which has a large variety of practical applications, rank and dense rank are typically used for ranking and scoring applications.

RANK and DENSE_RANK Functions in SQL Server 2005

SQL Server 2005 provides you with built-in RANK and DENSE_RANK functions that are similar to the ROW_NUMBER function. The difference between these functions and ROW_NUMBER is that, as I described earlier, ROW_NUMBER is not deterministic when the ORDER BY list is not unique. RANK and DENSE_RANK are always deterministic—that is, the same ranking values are assigned to rows with the same sort values. The difference between RANK and DENSE_RANK is that RANK might have gaps in the ranking values, but allows you to know how many rows have lower sort values. DENSE_RANK values have no gaps.

As an example, the following query (which produces the output shown in Table 4-31) returns both rank and dense rank values for the sales rows based on an ordering by quantity:

Table 4-31: Rank and Dense Rank

empid	qty	rnk	drnk
B	100	1	1
G	100	1	1
J	100	1	1
H	150	4	2
C	200	5	3
D	200	5	3
K	200	5	3
E	250	8	4
I	250	8	4
A	300	10	5
F	300	10	5

```
SELECT empid, qty,
  RANK() OVER(ORDER BY qty) AS rnk,
  DENSE_RANK() OVER(ORDER BY qty) AS drnk
FROM dbo.Sales
ORDER BY qty;
```

Here's a short quiz: what's the difference between the results of ROW_NUMBER, RANK and DENSE_RANK given a unique ORDER BY list?

For the answer, run the following code:

```
SELECT REVERSE('!gnihton yletulosbA');
```

Set-Based Solutions prior to SQL Server 2005

Rank and dense rank calculations using a set-based technique from pre–SQL Server 2005 are very similar to row number calculations. To calculate rank, use a subquery that counts the number of rows with a smaller sort value, and add one. To calculate dense rank, use a subquery that counts the distinct number of smaller sort values, and add one.

```
SELECT empid, qty,
  (SELECT COUNT(*) FROM dbo.Sales AS S2
   WHERE S2.qty < S1.qty) + 1 AS rnk,
  (SELECT COUNT(DISTINCT qty) FROM dbo.Sales AS S2
   WHERE S2.qty < S1.qty) + 1 AS drnk
FROM dbo.Sales AS S1
ORDER BY qty;
```

Of course, you can add a correlation to return partitioned calculations just like you did with row numbers.

NTILE

The NTILE function distributes rows into a specified number of tiles (or groups). The tiles are numbered 1 and on. Each row is assigned with the tile number to which it belongs. NTILE is based on row number calculation—namely, it is based on a requested order and can optionally be partitioned. Based on the number of rows in the table (or partition), the number of requested tiles, and the row number, you can determine the tile number for each row. For example, for a table with 10 rows, the value of NTILE(2) OVER (ORDER BY c) would be 1 for the first 5 rows in column c order, and 2 for the 6th through 10th rows.

Typically, NTILE calculations are used for analytical purposes such as calculating percentiles or arranging items in groups.

The task of "tiling" has more than one solution, and SQL Server 2005 implements a specific solution, called ANSI NTILE. I will describe the NTILE function implementation in SQL Server 2005, and then cover other solutions.

NTILE Function in SQL Server 2005

Calculating NTILE in SQL Server 2005 is as simple as calculating any of the other ranking values—in this case, using the NTILE function. The only difference is that NTILE accepts an input, the number of tiles, while the other ranking functions have no input. Because NTILE calculations are based on row numbers, NTILE has exactly the same issues regarding determinism that I described in the row numbers section.

As an example, the following query calculates NTILE values for the rows from the Sales table, producing three tiles, based on the order of *qty* and *empid*:

```
SELECT empid, qty,
  NTILE(3) OVER(ORDER BY qty, empid) AS tile
FROM dbo.Sales
ORDER BY qty, empid;
```

The output of this query is shown in Table 4-32.

Table 4-32: NTILE Query Output

empid	qty	tile
B	100	1
G	100	1
J	100	1
H	150	1
C	200	2
D	200	2
K	200	2
E	250	2
I	250	3
A	300	3
F	300	3

Note that when the number of tiles (*num_tiles*) does not evenly divide the count of rows in the table (*cnt*), the first *r* tiles (where *r* is *cnt % num_tiles*) get one more row than the others. In other words, the remainder is assigned to the first tiles first. In our example, the table has eleven rows, and three tiles were requested. The base tile size is 11/3 = 3 (integer

division). The remainder is 11 % 3 = 2. The % (modulo) operator provides the integer remainder after dividing the first integer by the second one. So the first two tiles get an additional row beyond the base tile size and end up with four rows.

As a more meaningful example, suppose you need to split the sales rows into three categories based on quantities: low, medium, and high. You want each category to have about the same number of rows. You can calculate NTILE(3) values based on *qty* order (using *empid* as a tiebreaker just to assure deterministic and reproducible results) and use a CASE expression to convert the tile numbers to more meaningful descriptions:

```
SELECT empid, qty,
  CASE NTILE(3) OVER(ORDER BY qty, empid)
    WHEN 1 THEN 'low'
    WHEN 2 THEN 'medium'
    WHEN 3 THEN 'high'
  END AS lvl
FROM dbo.Sales
ORDER BY qty, empid;
```

The output of this query is shown in Table 4-33.

Table 4-33: Descriptive Tiles

empid	qty	lvl
B	100	low
G	100	low
J	100	low
H	150	low
C	200	medium
D	200	medium
K	200	medium
E	250	medium
I	250	high
A	300	high
F	300	high

To calculate the range of quantities corresponding to each category as shown in Table 4-34, simply group the data by the tile number, returning the minimum and maximum sort values for each group:

Table 4-34: Ranges of Quantities Corresponding to Each Category

tile	lb	hb
1	100	150
2	200	250
3	250	300

```
WITH Tiles AS
(
  SELECT empid, qty,
    NTILE(3) OVER(ORDER BY qty, empid) AS tile
  FROM dbo.Sales
)
SELECT tile, MIN(qty) AS lb, MAX(qty) AS hb
FROM Tiles
GROUP BY tile
ORDER BY tile;
```

Other Set-Based Solutions to NTILE

Calculating ANSI NTILE prior to SQL Server 2005 using set-based techniques is more difficult, and obviously far more expensive. Calculating other types of tiling is not trivial even in SQL Server 2005.

The formula you use to calculate NTILE depends on what exactly you want to do with the remainder in case the number of rows in the table doesn't divide evenly by the number of tiles. You might want to use the ANSI NTILE function's approach, which says, "Just assign the remainder to the first tiles, one to each until it's all consumed." Another approach, which is probably more correct statistically is to more evenly distribute the remainder among the tiles instead of putting them into the initial tiles only.

I'll start with the latter approach, calculating NTILE values with even distribution, because it's simpler. You need two inputs to calculate the tile number for a row: the row number and the tile size. You already know how to calculate row numbers. To calculate the tile size, you divide the number of rows in the table by the requested number of tiles. The formula that calculates the target tile number is

```
(row_number - 1) / tile_size + 1
```

The trick that would allow you to distribute the remainder evenly is to use a decimal calculation when calculating the *tile_size* value, instead of an integer one. That is, instead of using an integer calculation of the tile size (*num_rows/num_tiles*), which will truncate the fraction, use *1.*numrows/numtiles*, which will return a more accurate decimal result. Finally, to get rid of the fraction in the tile number, convert the result back to an integer value.

Here's the complete query that produces tile numbers using the even-distribution approach and generates the output shown in Table 4-35:

Table 4-35: NTILE, Even Distribution of Remainder, 3 Tiles

empid	qty	tile
B	100	1
G	100	1
J	100	1
H	150	1
C	200	2
D	200	2
K	200	2
E	250	2
I	250	3
A	300	3
F	300	3

```
DECLARE @numtiles AS INT;
SET @numtiles = 3;

SELECT empid, qty,
  CAST((rn - 1) / tilesize + 1 AS INT) AS tile
FROM (SELECT empid, qty, rn,
        1.*numrows/@numtiles AS tilesize
      FROM (SELECT empid, qty,
              (SELECT COUNT(*) FROM dbo.Sales AS S2
               WHERE S2.qty < S1.qty
                  OR S2.qty = S1.qty
                     AND S2.empid <= S1.empid) AS rn,
              (SELECT COUNT(*) FROM dbo.Sales) AS numrows
            FROM dbo.Sales AS S1) AS D1) AS D2
ORDER BY qty, empid;
```

With three tiles, you can't see the even distribution of the remaining rows. If you run this code using nine tiles as input, you get the output shown in Table 4-36 where the even distribution is clearer.

Table 4-36: NTILE, Even Distribution of Remainder, 9 Tiles

empid	qty	tile
B	100	1
G	100	1
J	100	2
H	150	3
C	200	4
D	200	5
K	200	5
E	250	6
I	250	7
A	300	8
F	300	9

You can see in the result that the first tile contains two rows, the next three tiles contain one row each, the next tile contains two rows, and the last four tiles contain one row each. You can experiment with the input number of tiles to get a clearer picture of the even-distribution algorithm.

To get the same result as the ANSI NTILE function, where the remainder is distributed to the lowest-numbered tiles, you need a different formula. First, the calculations involve only integers. The inputs you need for the formula in this case include the row number, tile size, and remainder (*number of rows in the table % number of requested tiles*). These inputs are used in calculating NTILE with non-even distribution.

The formula for the target tile number is as follows:

```
if row_number <= (tilesize + 1) * remainder then
  tile_number = (row_number - 1) / (tile_size + 1) + 1
else
  tile_number = (row_number - remainder - 1) / tile_size + 1
```

Translated to T-SQL, the query (which produces the output shown in Table 4-37) looks like this:

Table 4-37: NTILE, Remainder Added to First Groups

empid	qty	tile
B	100	1
G	100	1
J	100	2
H	150	2
C	200	3
D	200	4
K	200	5
E	250	6
I	250	7
A	300	8
F	300	9

```
DECLARE @numtiles AS INT;
SET @numtiles = 9;

SELECT empid, qty,
  CASE
    WHEN rn <= (tilesize+1) * remainder
      THEN (rn-1) / (tilesize+1) + 1
    ELSE (rn - remainder - 1) / tilesize + 1
    END AS tile

FROM (SELECT empid, qty, rn,
        numrows/@numtiles AS tilesize,
        numrows%@numtiles AS remainder
      FROM(SELECT empid, qty,
             (SELECT COUNT(*) FROM dbo.Sales AS S2
              WHERE S2.qty < S1.qty
                OR S2.qty = S1.qty
                    AND S2.empid <= S1.empid) AS rn,
            (SELECT COUNT(*) FROM dbo.Sales) AS numrows
           FROM dbo.Sales AS S1) AS D1) AS D2
ORDER BY qty, empid;
```

The output is the same as the one you would get using the SQL Server 2005 NTILE function; the first tiles get an additional row until the remainder is consumed.

Auxiliary Table of Numbers

An auxiliary table of numbers is a very powerful tool that I often use in my solutions. So I decided to dedicate a section in this chapter to it. In this section, I'll simply describe the concept and the methods used to generate such a table. I'll refer to this auxiliary table throughout the book and demonstrate many of its applications.

An auxiliary table of numbers (call it Nums) is simply a table that contains the integers between 1 and N for some (typically large) value of N. I recommend that you create a permanent Nums table and populate it with as many values as you might need for your solutions.

The code in Listing 4-8 demonstrates how to create such a table containing 1,000,000 rows. Of course, you might want a different number of rows, depending on your needs.

Listing 4-8: Creating and populating auxiliary table of numbers

```
SET NOCOUNT ON;
USE AdventureWorks;
GO
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;
```

Tip Because there are so many practical uses for a Nums table, you'll probably end up needing to access it from various databases. To avoid the need to refer to it using the fully qualified name AdventureWorks.dbo.Nums, you can create a synonym in the model database pointing to Nums in AdventureWorks like this:

```
USE model;
CREATE SYNONYM dbo.Nums FOR AdventureWorks.dbo.Nums;
```

Creating the synonym in *model* will make it available in all newly created databases from that point on, including tempdb after SQL Server is restarted. For existing databases, you will just need to explicitly run the CREATE SYNONYM command once.

In practice, it doesn't really matter how you populate the Nums table because you run this process only once. Nevertheless, I used an optimized process that populates the table in a very fast manner. The process demonstrates the technique of creating Nums with a multiplying INSERT loop.

The code keeps track of the number of rows already inserted into the table in a variable called *@rc.* It first inserts into Nums the row where *n = 1.* It then enters a loop while *@rc * 2* <= *@max* (@ max is the desired number of rows). In each iteration, the process inserts into Nums the result of a query that selects all rows from Nums after adding *@rc* to each *n* value. This technique doubles the number of rows in Nums in each iteration—that is, first {1} is inserted, then {2}, then {3, 4}, then {5, 6, 7, 8}, then {9, 10, 11, 12, 13, 14, 15, 16}, and so on.

As soon as the table is populated with more than half the target number of rows, the loop ends. Another INSERT statement after the loop inserts the remaining rows using the same INSERT statement as within the loop, but this time with a filter to ensures that only values <= *@max* will be loaded.

The main reason that this process runs fast is that it minimizes writes to the transaction log compared to other available solutions. This is achieved by minimizing the number of INSERT statements (the number of INSERT statements is *CEILING(LOG2(@max)) + 1*). This code populated the Nums table with 1,000,000 rows in 6 seconds on my laptop. As an exercise, you can try populating the Nums table using a simple loop of individual inserts and see how long it takes.

Whenever you need the first *@n* numbers from Nums, simply query it, specifying *WHERE n*<= *@n* as the filter. An index on the *n* column will ensure that the query will scan only the required rows and no others.

If you're not allowed to add permanent tables in the database, you can create a table-valued UDF with a parameter for the number of rows needed. You use the same logic as used above to generate the required number of values.

In SQL Server 2005, you can use the new recursive CTEs and ROW_NUMBER function to create extremely efficient solutions that generate a table of numbers on the fly.

I'll start with a naïve solution that is fairly slow (about 20 seconds, with results discarded). The following solution uses a simple recursive CTE, where the anchor member generates a row with *n = 1*, and the recursive member adds a row in each iteration with *n = prev n + 1*:

```
DECLARE @n AS BIGINT;
SET @n = 1000000;

WITH Nums AS
(
  SELECT 1 AS n
  UNION ALL
  SELECT n + 1 FROM Nums WHERE n < @n
)
SELECT n FROM Nums
OPTION(MAXRECURSION 0);
```

Note If you're running the code to test it, remember to turn on the Discard Results After Execution option in SSMS; otherwise, you will get an output with a million rows.

You have to use a hint that removes the default recursion limit of 100. This solution runs for about 20 seconds.

You can optimize the solution significantly by using a CTE (call it *Base*) that generates as many rows as the square root of the target number of rows. Take the cross join of two instances of *Base* to get the target number of rows, and finally, generate row numbers for the result to serve as the sequence of numbers.

Here's the code that implements this approach:

```
DECLARE @n AS BIGINT;
SET @n = 1000000;
```

```
WITH Base AS
(
  SELECT 1 AS n
  UNION ALL
  SELECT n + 1 FROM Base WHERE n < CEILING(SQRT(@n))
),
Expand AS
(
  SELECT 1 AS c
  FROM Base AS B1, Base AS B2
),
Nums AS
(
  SELECT ROW_NUMBER() OVER(ORDER BY c) AS n
  FROM Expand
)
SELECT n FROM Nums WHERE n <= @n
OPTION(MAXRECURSION 0);
```

This solution runs for only 0.8 seconds (results discarded).

Next, I'll describe the third approach to generate Nums. You start with a CTE that has only two rows, and multiply the number of rows with each following CTE by cross-joining two instances of the previous CTE. With *n* levels of CTEs (0-based), you reach 2^{2n} rows. For example, with 5 levels, you get 4,294,967,296 rows.

Another CTE generates row numbers, and finally the outer query filters the desired number of values (where *row number column* <= *input*). Remember that when you filter a *row number* <= *some* value, SQL Server doesn't bother to generate row numbers beyond that point. So you shouldn't be concerned about performance. It's not the case that your code will really generate more than four billion rows every time and then filter.

Here's the code that implements this approach:

```
DECLARE @n AS BIGINT;
SET @n = 1000000;

WITH
L0   AS(SELECT 1 AS c UNION ALL SELECT 1),
L1   AS(SELECT 1 AS c FROM L0 AS A, L0 AS B),
L2   AS(SELECT 1 AS c FROM L1 AS A, L1 AS B),
L3   AS(SELECT 1 AS c FROM L2 AS A, L2 AS B),
L4   AS(SELECT 1 AS c FROM L3 AS A, L3 AS B),
L5   AS(SELECT 1 AS c FROM L4 AS A, L4 AS B),
Nums AS(SELECT ROW_NUMBER() OVER(ORDER BY c) AS n FROM L5)
SELECT n FROM Nums WHERE n <= @n;
```

It runs for about 0.9 seconds to generate a sequence of 1,000,000 numbers.

As I mentioned earlier, you can wrap the logic in a UDF. The value of this solution is that it will never near the MAXRECURSION limit, and that limit cannot be modified in a UDF definition. The code in Listing 4-9 encapsulates the last solution's logic in a UDF.

Listing 4-9: UDF returning an auxiliary table of numbers

```
CREATE FUNCTION dbo.fn_nums(@n AS BIGINT) RETURNS TABLE
AS
RETURN
  WITH
  L0   AS(SELECT 1 AS c UNION ALL SELECT 1),
  L1   AS(SELECT 1 AS c FROM L0 AS A, L0 AS B),
  L2   AS(SELECT 1 AS c FROM L1 AS A, L1 AS B),
  L3   AS(SELECT 1 AS c FROM L2 AS A, L2 AS B),
  L4   AS(SELECT 1 AS c FROM L3 AS A, L3 AS B),
  L5   AS(SELECT 1 AS c FROM L4 AS A, L4 AS B),
  Nums AS(SELECT ROW_NUMBER() OVER(ORDER BY c) AS n FROM L5)
  SELECT n FROM Nums WHERE n <= @n;
GO
```

To test the function, run the following code, which returns an auxiliary table with 10 numbers:

```
SELECT * FROM dbo.fn_nums(10) AS F;
```

Existing and Missing Ranges (Also Known as Islands and Gaps)

To put your knowledge of subqueries, table expressions, and ranking calculations into action, I'll provide a couple of problems that have many applications in production environments. I'll present a generic form of the problem, though, so you can focus on the techniques and not the data.

Create and populate a table named T1 by running the code in Listing 4-10.

Listing 4-10: Creating and populating the T1 table

```
USE tempdb;
GO
IF OBJECT_ID('dbo.T1') IS NOT NULL
  DROP TABLE dbo.T1
GO
CREATE TABLE dbo.T1(col1 INT NOT NULL PRIMARY KEY);
INSERT INTO dbo.T1(col1) VALUES(1);
INSERT INTO dbo.T1(col1) VALUES(2);
INSERT INTO dbo.T1(col1) VALUES(3);
INSERT INTO dbo.T1(col1) VALUES(100);
INSERT INTO dbo.T1(col1) VALUES(101);
INSERT INTO dbo.T1(col1) VALUES(103);
INSERT INTO dbo.T1(col1) VALUES(104);
INSERT INTO dbo.T1(col1) VALUES(105);
INSERT INTO dbo.T1(col1) VALUES(106);
```

You have two tasks. The first task is to return the missing ranges of keys within the data, generating the output shown in Table 4-38.

Table 4-38: Missing Ranges

start_range	end_range
4	99
102	102

The second task is to return ranges of consecutive keys in the data, generating the output shown in Table 4-39.

Table 4-39: Existing Ranges

start_range	end_range
1	3
100	101
103	106

These problems manifest in production systems in many forms—for example, availability or nonavailability reports. In some cases, the values appear as integers, such as in our case. In other cases, they appear as *datetime* values. The techniques that are needed to solve the problem with integers apply to *datetime* values with very minor revisions.

Missing Ranges (Also Known as Gaps)

There are several approaches you can use to solve the problem of missing ranges (gaps), but first it's important to identify the steps in the solution before you start coding.

One approach can be described by the following steps:

- Find the points before the gaps, and add one to each.

- For each starting point of a gap, find the next existing value in the table and subtract one.

Having the logical aspects of the steps resolved, you can start coding. You will find in the preceding logical steps that the chapter covered all the fundamental techniques that are mentioned—namely, finding "points before gaps" and finding the "next" existing value.

The following query returns the points before the gaps shown in Table 4-40:

Table 4-40: Points Before Gaps

col1
3
101
106

```
SELECT col1
FROM dbo.T1 AS A
WHERE NOT EXISTS
  (SELECT * FROM dbo.T1 AS B
   WHERE B.col1 = A.col1 + 1);
```

Remember a point before a gap is a value after which the next doesn't exist.

Notice in the output that the last row is of no interest to us because it's before infinity. The following query returns the starting points of the gaps, and its output is shown in Table 4-41. It achieves this by adding one to the points before the gaps to get the first values in the gaps, filtering out the point before infinity.

Table 4-41: Starting Points of Gaps

start_range
4
102

```
SELECT col1 + 1 AS start_range
FROM dbo.T1 AS A
WHERE NOT EXISTS
  (SELECT * FROM dbo.T1 AS B
   WHERE B.col1 = A.col1 + 1)
  AND col1 < (SELECT MAX(col1) FROM dbo.T1);
```

Finally, for each starting point in the gap, you use a subquery to return the next value in the table minus 1—in other words, the end of the gap:

```
SELECT col1 + 1 AS start_range,
  (SELECT MIN(col1) FROM dbo.T1 AS B
   WHERE B.col1 > A.col1) - 1 AS end_range
FROM dbo.T1 AS A
WHERE NOT EXISTS
  (SELECT * FROM dbo.T1 AS B
   WHERE B.col1 = A.col1 + 1)
  AND col1 < (SELECT MAX(col1) FROM dbo.T1);
```

That's one approach to solving the problem. Another approach, which I find much simpler and more intuitive is the following one:

- To each existing value, match the next existing value, generating current, next pairs.
- Keep only pairs where next minus current is greater than one.

- With the remaining pairs, add one to the current and subtract one from the next.

This approach relies on the fact that adjacent values with a distance greater than one represent the boundaries of a gap. Identifying a gap based on identification of the next existing value is another useful technique.

To translate the preceding steps to T-SQL, the following query simply returns the next value for each current value, generating the output shown in Table 4-42:

```
SELECT col1 AS cur
  (SELECT MIN(col1) FROM dbo.T1 AS B
   WHERE B.col1 > A.col1) AS nxt
FROM dbo.T1 AS A;
```

Table 4-42: Current, Next Pairs

cur	nxt
1	2
2	3
3	100
100	101
101	103
103	104
104	105
105	106
106	NULL

Finally, you create a derived table out of the previous step's query, and you keep only pairs where *nxt*—*cur* is greater than one. You add one to *cur* to get the actual start of the gap, and subtract one from *nxt* to get the actual end of the gap:

```
SELECT cur + 1 AS start_range, nxt - 1 AS end_range
FROM (SELECT col1 AS cur,
        (SELECT MIN(col1) FROM dbo.T1 AS B
        WHERE B.col1 > A.col1) AS nxt
      FROM dbo.T1 AS A) AS D
WHERE nxt - cur > 1;
```

Note that this solution got rid of the point before infinity with no special treatment, because the *nxt* value for it was NULL.

I compared the performance of the two solutions and found them to be similar. However, the second solution is doubtless simpler and more intuitive, and that's a big advantage in terms of readability and maintenance.

By the way, if you need to return the list of individual missing values as opposed to missing ranges, using the Nums table, the task is very simple:

```
SELECT n FROM dbo.Nums
WHERE n BETWEEN (SELECT MIN(col1) FROM dbo.T1)
            AND (SELECT MAX(col1) FROM dbo.T1)
  AND NOT EXISTS(SELECT * FROM dbo.T1 WHERE col1 = n);
```

Existing Ranges (Also Known as Islands)

Returning ranges of existing values or collapsing ranges with consecutive values involves a concept I haven't discussed yet—a grouping factor. You basically need to group data by a factor that does not exist in the data as a base attribute. In our case, you need to calculate some *x* value for all members of the first subset of consecutive values {1, 2, 3}, some *y* value for the second {100, 101}, some *z* value for the third {104, 105, 106}, and so on. Once you have this grouping factor available, you can group the data by this factor and return the minimum and maximum *col1* values in each group.

One approach to calculating this grouping factor brings me to another technique: calculating the *min* or *max* value of a group of consecutive values. Take the group {1, 2, 3} as an example. If you will manage to calculate for each of the

members the *max* value in the group (3), you will be able to use it as your grouping factor.

The logic behind the technique to calculating the maximum within a group of consecutive values is: return the minimum value that is greater than or equal to the current, after which there's a gap. Here's the translation to T-SQL, yielding the output shown in Table 4-43:

```
SELECT col1
  (SELECT MIN(col1) FROM dbo.T1 AS B
   WHERE B.col1 >= A.col1
     AND NOT EXISTS
       (SELECT * FROM dbo.T1 AS C
        WHERE B.col1 = C.col1 - 1)) AS grp
FROM dbo.T1 AS A;
```

Table 4-43: Grouping Factor

col1	grp
1	3
2	3
3	3
100	101
101	101
103	106
104	106
105	106
106	106

The rest is really easy: create a derived table out of the previous step's query, group the data by the grouping factor, and return the minimum and maximum values for each group:

```
SELECT MIN(col1) AS start_range, MAX(col1) AS end_range
FROM (SELECT col1,
        (SELECT MIN(col1) FROM dbo.T1 AS B
        WHERE B.col1 >= A.col1
          AND NOT EXISTS
            (SELECT * FROM dbo.T1 AS C
              WHERE B.col1 = C.col1 - 1)) AS grp
     FROM dbo.T1 AS A) AS D
GROUP BY grp;
```

This solution solves the problem, but I'm not sure I'd qualify it as a very simple and intuitive solution with satisfactory performance.

In search for a simpler and faster way to calculate the grouping factor, check out the output shown in Table 4-44 of the following query that simply produces row numbers based on *col1* order.

Table 4-44: Row Numbers Based on *col1* Order

col1	rn
1	1
2	2
3	3
100	4
101	5

103	6
104	7
105	8
106	9

```
SELECT col1, ROW_NUMBER() OVER(ORDER BY col1) AS rn
FROM dbo.T1;
```

If you're working with SQL Server 2000, you'd probably want to use the IDENTITY-based technique I described earlier for calculating row numbers.

See if you can identify a relationship between the way the *col1* values increment and the way row numbers do.

Well, their difference remains constant within the same group of consecutive values, because it has no gaps. As soon as you get to a new group, the difference between *col1* and the row number increases.

To make this idea more tangible, calculate the difference between *col1* and the row number and examine the result shown in Table 4-45. This key technique shows one way to calculate the group factor with ROW_NUMBER.

Table 4-45: Difference between *col1* and Row Number

col1	diff
1	0
2	0
3	0
100	96
101	96
103	97
104	97
105	97
106	97

```
SELECT col1, col1 - ROW_NUMBER() OVER(ORDER BY col1) AS diff
FROM dbo.T1;
```

Now it's crystal clear. You have created a grouping factor with close to zero effort. Simple and fast!

Now simply replace in the previous solution the derived table query with the preceding query to get the desired result:

```
SELECT MIN(col1) AS start_range, MAX(col1) AS end_range
FROM (SELECT col1, col1 - ROW_NUMBER() OVER(ORDER BY col1) AS grp
      FROM dbo.T1) AS D
GROUP BY grp;
```

Conclusion

This chapter covered many subjects, all related to subqueries. I discussed scalar and list subqueries, self-contained and correlated subqueries, table expressions, and ranking calculations.

It's important to make mental notes of the fundamental techniques that I point out here and throughout the book, such as generating duplicates using an auxiliary table of numbers, introducing a tiebreaker, finding points before gaps, returning the next or previous value, calculating a grouping factor, and so on. This builds your T-SQL vocabulary and enhances your skills. As you progress with this approach, you'll see that it becomes easier and easier to identify fundamental elements in a problem. Having already resolved and polished key techniques separately in a focused manner, you will use them naturally to solve problems.