

Chapters *To Go*



Inside Microsoft SQL Server 2005: T-SQL Querying

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

Reprinted for Elango Sugumaran, IBM

esugumar@in.ibm.com

Reprinted with permission as a subscription benefit of **Books24x7**,
<http://www.books24x7.com/>

All rights reserved. Reproduction and/or distribution in whole or in part in electronic, paper or other forms without written permission is prohibited.



Chapter 5: Joins and Set Operations

This chapter covers joins and set operations—their logical aspects as well as their physical/ performance aspects. I'll demonstrate practical applications for each type of join and set operation. I have used the ANSI SQL terminology to categorize the elements of the language that I'll cover here. *Joins* (CROSS, INNER, OUTER) refer to *horizontal* operations (loosely speaking) between tables, while *set operations* (UNION, EXCEPT, INTERSECT) refer to *vertical* operations between tables.

Joins

Joins are operations that allow you to match rows between tables. I informally referred to these operations as horizontal ones because the virtual table resulting from a join operation between two tables contains all columns from both tables.

I'll first describe the different syntaxes for joins supported by the standard, and I'll also mention the proprietary elements in T-SQL. I'll then describe the fundamental join types and their applications followed by other categorizations of joins. I'll also have a focused discussion on the internal processing of joins—namely, join algorithms.

You'll have a couple of chances to practice what you've learned by trying to solve a problem that encompasses previously discussed aspects of joins.

Old Style vs. New Style

T-SQL supports two different syntaxes for joins. There's a lot of confusion surrounding the two. When do you use each? Which performs better? Which is standard and which is proprietary? Is the older syntax going to be deprecated soon? And so on. Hopefully, this chapter will clear the fog.

I'll start by saying there are two different syntaxes for joins supported by the ANSI standard, and neither is in a process of deprecation yet. The join elements of the older standard are a complete part of the newer. This means that you can use either one without worrying that it will not be supported by Microsoft SQL Server sometime soon. SQL Server will not remove support for implemented features that were not deprecated by the standard.

The older of the two syntaxes was introduced in ANSI SQL:1989. What distinguishes it from the newer one is the use of commas to separate table names that appear in the FROM clause, and the absence of the JOIN keyword and the ON clause:

```
FROM T1, T2
WHERE where_filter
```

The ANSI SQL:1989 syntax had support only for cross and inner join types. It did not have support for outer joins.

The newer syntax was introduced in ANSI SQL:1992, and what distinguishes it from the older syntax is the removal of the commas and the introduction of the JOIN keyword and the ON clause:

```
FROM T1 <join_type> JOIN T2 ON <on_filter>
WHERE where_filter
```

ANSI SQL:1992 introduced support for outer joins, and this drove the need for a separation of filters—the ON filter and the WHERE filter. I'll explain this in detail in the outer joins section.

Part of the confusion surrounding the two syntaxes has to do with the fact that T-SQL supported a proprietary syntax for outer joins before SQL Server added support for the ANSI SQL:1992 syntax. There was a practical need for outer joins, and SQL Server provided an answer to that need. Particularly, I'm talking about the old-style proprietary outer join syntax using *= and =* for left outer and right outer joins, respectively.

For backward compatibility reasons, SQL Server has not removed support for the proprietary outer join syntax thus far. However, these syntax elements were deprecated in SQL Server 2005 and will work only under a backward-compatibility flag. All other join syntax elements are standard and are not being considered for deprecation—neither by the standard, nor by SQL Server.

As I describe the different fundamental join types, I'll discuss both syntaxes and give you my opinion regarding which one I find more convenient to use and why.

Fundamental Join Types

As I describe the different fundamental join types—cross, inner, and outer—keep in mind the phases in query logical processing that I described in detail in the Chapter 1. In particular, keep in mind the logical phases involved in join processing.

Each fundamental join type takes place only between two tables. Even if you have more than two tables in the FROM clause, the first three query logical processing phases will take place between two tables at a time. Each join will result in a virtual table, which in turn will be joined with the next table in the FROM clause. This process will continue until all tables in the FROM clause are processed.

The fundamental join types differ in the logical phases that they apply. Cross join applies only the first (Cartesian product), inner join applies the first and the second (Cartesian product, ON filter), and outer join applies all three (Cartesian product, ON filter, add outer rows).

CROSS

A cross join performs a Cartesian product between two tables. In other words, it returns a row for each possible combination of a row from the left table and a row from the right table. If the left table has n rows and the right table has m rows, a cross join will return a table with $n \times m$ rows.

There are many practical applications to cross joins, but I'll start with a very simple example— a plain cross.

The following query produces all possible pairs of employees from the Employees table in the Northwind database:

```
USE Northwind;

SELECT E1.FirstName, E1.LastName AS emp 1,
       E2.FirstName, E2.LastName AS emp 2
FROM dbo.Employees AS E1
     CROSS JOIN dbo.Employees AS E2;
```

Because the Employees table contains nine rows, the result set will contain 81 rows.

And here's the ANSI SQL:1989 syntax you would use for the same task:

```
SELECT E1.FirstName, E1.LastName AS emp1,
       E2.FirstName, E2.LastName AS emp2
FROM dbo.Employees AS E1,  dbo.Employees AS E2;
```

For cross joins only, I prefer using a comma (as opposed to using the CROSS JOIN keywords) because it allows for shorter code. I also find the older syntax to be more natural and readable. The optimizer will produce the same plan for both, so you shouldn't have any concerns about performance.

As you will see later on, I will give a different recommendation for inner joins. Now let's look at more sophisticated uses of cross joins.

In Chapter 4, I presented a powerful key technique to generate duplicates. Recall that I used an auxiliary table of numbers (Nums) as follows to generate the requested number of duplicates of each row:

```
SELECT ...
FROM T1, Nums
WHERE n <= <num_of_dups>
```

The preceding technique will generate in the result set *num_of_dups* duplicates of each row in T1. As a practical example, suppose you need to fill an Orders table with sample data for testing. You have a Customers table with sample customer information and an Employees table with sample employee information. You want to generate, for each combination of a customer and an employee, an order for each day in January 2006.

I will demonstrate this technique, generating test data based on duplicates, in the Northwind database. The Customers table contains 91 rows, the Employees table contains 9 rows, and for each customer-employee combination, you need an order for each day in January 2006—that is, for 31 days. The result set should contain 25,389 rows ($91 \times 9 \times 31 = 25,389$). Naturally, you will want to store the result set in a target table and generate an order ID for each order.

You already have tables with customers and employees, but there's a missing table—you need a table to represent the days. You probably guessed already that the Nums table will assume the role of the missing table:

```
SELECT CustomerID, EmployeeID,
       DATEADD(day, n-1, '20060101') AS OrderDate
```

```
FROM dbo.Customers, dbo.Employees, dbo.Nums
WHERE n <= 31;
```

You cross Customers, Employees, and Nums, filtering the first 31 values of *n* from the Nums table for the 31 days of the month. In the SELECT list, you calculate the specific target dates by adding *n* – 1 days to the first date of the month, January 1, 2006.

The last missing element is the order ID. But you can easily generate it using the ROW_NUMBER function in SQL Server 2005, or the IDENTITY function or property in SQL Server 2000.

In practice, you'd probably want to encapsulate this logic in a stored procedure that accepts the date range as input. Instead of using a literal for the number of days in the filter, you will use the following expression:

```
DATEDIFF(day, @fromdate, @todate) + 1
```

Similarly, the DATEADD function in the SELECT list will refer to @fromdate instead of a literal base date:

```
DATEADD(day, n-1, @fromdate) AS OrderDate
```

Here's the code that you would need in SQL Server 2000 to generate the test data to populate a target table. I'm using the IDENTITY function to generate the order IDs and the date range boundaries as input arguments:

```
DECLARE @fromdate AS DATETIME, @todate AS DATETIME;
SET @fromdate = '20060101';
SET @todate = '20060131';
```

```
SELECT IDENTITY(int,1, 1) AS OrderID,
       CustomerID, EmployeeID,
       DATEADD(day, n-1, @fromdate) AS OrderDate
```

```
INTO dbo.MyOrders
FROM dbo.Customers, dbo.Employees, dbo.Nums
WHERE n <= DATEDIFF(day, @fromdate, @todate) + 1;
```

In SQL Server 2005, you can use the ROW_NUMBER function instead of the IDENTITY function, and assign the row numbers based on a desired ordering column (for example, *OrderDate*):

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

```
DECLARE @fromdate AS DATETIME, @todate AS DATETIME;
SET @fromdate = '20060101';
SET @todate = '20060131';
```

```
WITH Orders
AS
(
    SELECT CustomerID, EmployeeID,
           DATEADD(day, n-1, @fromdate) AS OrderDate
    FROM dbo.Customers, dbo.Employees, dbo.Nums
    WHERE n >= DATEDIFF(day, @fromdate, @todate) + 1
)
SELECT ROW_NUMBER() OVER (ORDER BY OrderDate) AS OrderID,
       CustomerID, EmployeeID, OrderDate
INTO dbo.MyOrders
FROM Orders;
```

When you're done experimenting with this code, don't forget to drop the MyOrders table:

```
DROP TABLE dbo.MyOrders;
```

Another application of cross joins allows you to improve performance of queries that apply calculations between row attributes and aggregates over rows. To demonstrate this fundamental key technique, I'll use the sales table in the *pubs* database. First, create an index on the *qty* column, which is important for our task:

```
USE pubs;
CREATE INDEX idx_qty ON dbo.sales(qty);
```

The task at hand is to calculate for each sale that sale's percentage of total quantity sold, and the difference between the sale quantity and the average quantity for all sales. The intuitive way for programmers to write calculations between row

attributes and aggregates over rows is to use subqueries. The following code (which produces the output shown in [Table 5-1](#)) demonstrates the subquery approach:

Table 5-1: Sales Information, Including Percentage of Total and Difference from Average

stor_id	ord_num	title_id	ord_date	qty	per	diff
6380	6871	BU1032	1994-09-14	5	1.01	-18
6380	722a	PS2091	1994-09-13	3	0.61	-20
7066	A2976	PC8888	1993-05-24	50	10.14	27
7066	QA7442.3	PS2091	1994-09-13	75	15.21	52
7067	D4482	PS2091	1994-09-14	10	2.03	-13
7067	P2121	TC3218	1992-06-15	40	8.11	17
7067	P2121	TC4203	1992-06-15	20	4.06	-3
7067	P2121	TC7777	1992-06-15	20	4.06	-3
7131	N914008	PS2091	1994-09-14	20	4.06	-3
7131	N914014	MC3021	1994-09-14	25	5.07	2
7131	P3087a	PS1372	1993-05-29	20	4.06	-3
7131	P3087a	PS2106	1993-05-29	25	5.07	2
7131	P3087a	PS3333	1993-05-29	15	3.04	-8
7131	P3087a	PS7777	1993-05-29	25	5.07	2
7896	QQ2299	BU7832	1993-10-28	15	3.04	-8
7896	TQ456	MC2222	1993-12-12	10	2.03	-13
7896	X999	BU2075	1993-02-21	35	7.10	12
8042	423LL922	MC3021	1994-09-14	15	3.04	-8
8042	423LL930	BU1032	1994-09-14	10	2.03	-13
8042	P723	BU1111	1993-03-11	25	5.07	2
8042	QA879.1	PC1035	1993-05-22	30	6.09	7

```
SELECT stor_id, ord_num, title_id,
       CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
       CAST(1.*qty / (SELECT SUM(qty) FROM dbo.sales) * 100
            AS DECIMAL(5, 2)) AS per,
       qty - (SELECT AVG(qty) FROM dbo.sales) AS diff
FROM dbo.sales;
```

Before I get into the performance aspects of the query, I first want to discuss some of its logical aspects. As you can see, both the total quantity and the average quantity are obtained using self-contained subqueries. The expression calculating the percentage takes into account the way expressions are processed in T-SQL. The datatype of an expression is determined by the datatype with the higher precedence among the participating operands. This means that *qty / totalqty* (where *totalqty* stands for the subquery returning the total quantity) will yield an integer result—because both operands are integers, and the */* operator will be integer division. Because *qty* is a portion of *totalqty*, you'd always get a zero as the result, because the remainder is truncated. To get an accurate decimal calculation, you need to convert the operands to decimals. This can be achieved by using either explicit conversion or implicit conversion, as I did. *1.*qty* will implicitly convert the *qty* value to a decimal because a decimal is higher in precedence than an integer. Consequently, *totalqty* will also be converted to a decimal, and so will the integer literal 100. Finally, I used an explicit conversion to DECIMAL(5, 2) for the *per* result column to maintain an accuracy of only two decimal places. Three digits to the left of the decimal point are sufficient for the percentages here because the highest possible value is 100.

As for performance, examine this query's execution plan, which is shown in [Figure 5-1](#).

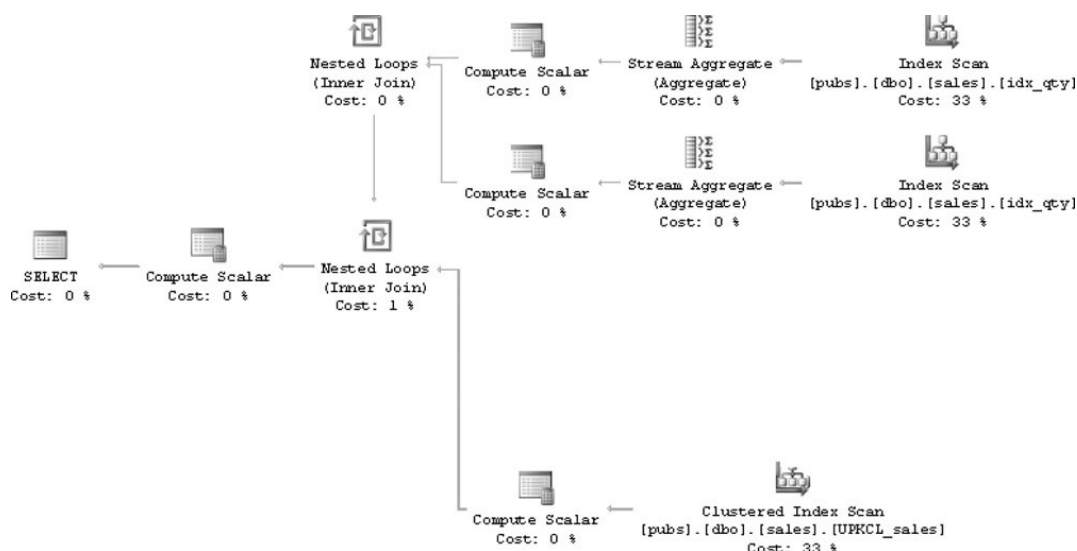


Figure 5-1: Execution plan for obtaining aggregates with subqueries

You will notice that the index I created on the *qty* column is scanned twice—once to calculate the sum, and once to calculate the average. In other words, provided that you have an index on the aggregated column, the index will be scanned once for each subquery that returns an aggregate. If you don't have an index containing the aggregated column, matters are even worse, as you'll get a table scan for each subquery.

This query can be optimized using a key technique that utilizes a cross join. You can calculate all needed aggregates in one query that will require only a single index or table scan. Such a query will produce a single result row with all aggregates. You create a derived table defined by this query and cross it with the base table. Now you have access to both attributes and aggregates. Here's the solution query, which produces the more optimal plan shown in [Figure 5-2](#):

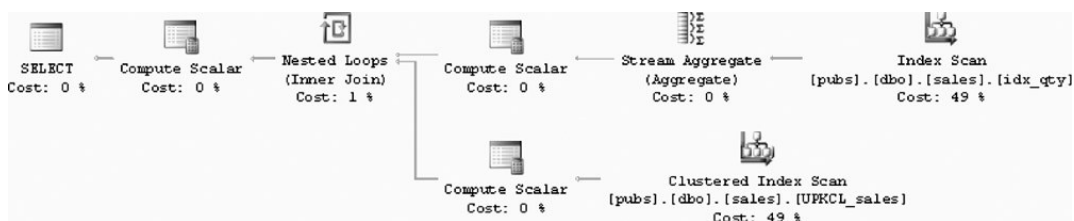


Figure 5-2: Execution plan for obtaining aggregates with a cross join

```
SELECT stor_id, ord_num, title_id,
       CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
       CAST(1.*qty / sumqty * 100 AS DECIMAL(5, 2)) AS per,
       qty - avgqty AS diff
FROM dbo.sales,
     (SELECT SUM(qty) AS sumqty, AVG(qty) AS avgqty
      FROM dbo.sales) AS AGG;
```

As you can see in the plan, the index on the *qty* column is scanned only once, and both aggregates are calculated with the same scan. Of course, in SQL Server 2005 you can use a common table expression (CTE), which you might find easier to read:

```
WITH Agg AS
(
    SELECT SUM(qty) AS sumqty, AVG(qty) AS avgqty
    FROM dbo.sales
)
SELECT stor_id, ord_num, title_id,
       CONVERT(VARCHAR(10), ord_date, 120) AS ord_date, qty,
       CAST(1.*qty / sumqty * 100 AS DECIMAL(5,2)) AS per,
       qty - avgqty AS diff
FROM dbo.sales, Agg;
```

You will find that both queries generate the same plan. In Chapter 6, I'll demonstrate how to use the new *OVER* clause in SQL Server 2005 to tackle similar problems.

Once you're done experimenting with this technique, drop the index on the *qty* column:

```
DROP INDEX dbo.sales.idx_qty;
```

INNER

Inner joins are used to match rows between two tables based on some criterion. Out of the first three query logical processing phases, inner joins apply the first two—namely, Cartesian product and ON filter. There's no phase that adds outer rows. Consequently, if an INNER JOIN query contains both an ON clause and a WHERE clause, logically they are applied one after the other. With one exception, there's no difference between specifying a logical expression in the ON clause or in the WHERE clause of an INNER JOIN, because there's no intermediate step that adds outer rows between the two.

The one exception is when you specify GROUP BY ALL. Remember that GROUP BY ALL adds back groups that were filtered out by the WHERE clause, but it does not add back groups that were filtered out by the ON clause. Remember also that this is a nonstandard legacy feature that you should avoid using.

As for performance, when not using the GROUP BY ALL option, you will typically get the same plan regardless of where you place the filter expression. That's because the optimizer is aware that there's no difference. I should always be cautious when saying such things related to optimization choices because the process is so dynamic.

As for the two supported join syntaxes, using the ANSI SQL:1992 syntax, you have more flexibility in choosing which clause you will use to specify a filter expression. Because logically it makes no difference where you place your filters, and typically there's also no performance difference, your guideline should be natural and intuitive writing. Write in a way that feels more natural to you and to the programmers who need to maintain your code. For example, to me a filter that matches attributes between the tables should appear in the ON clause, while a filter on an attribute from only one table should appear in the WHERE clause. I'll use the following query to return orders placed by U.S. customers:

```
USE Northwind;

SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C
     JOIN dbo.Orders AS O
       ON C.CustomerID= O.CustomerID
WHERE Country = 'USA';
```

Using the ANSI SQL:1989 syntax, you have no choice but to specify all filter expressions in the WHERE clause:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C,  dbo.Orders AS O
WHERE C.CustomerID = O.CustomerID
     AND Country = 'USA';
```

Remember that the discussion here is about inner joins; with outer joins, there are logical differences between specifying a filter expression in the ON clause and specifying it in the WHERE clause.

I mentioned earlier that I like using the ANSI SQL:1989 syntax for cross joins. However, with inner joins, my recommendation is different. The reason for this is that there's a risk in using the ANSI SQL:1989 syntax. If you forget to specify the join condition, unintentionally you get a cross join, as demonstrated in the following code:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C,  dbo.Orders AS O;
```

In SQL Server Management Studio (SSMS), the query plan for a cross join will include a join operator marked with a yellow warning symbol, and the pop-up details will say "No Join Predicate" in the Warnings section. This warning is designed to alert you that you might have forgotten to specify a join predicate.

However, if you explicitly specify INNER JOIN when you write an inner join query, an ON clause is required. If you forget to specify any join condition, the parser traps the error and the query is not run:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C JOIN dbo.Orders AS O;
```

```
Msg 102, Level 15, State 1, Line 2
Incorrect syntax near ';'.
```

The parser finds a semicolon after *dbo.Orders AS O*, even though it expects something else (an ON clause or other

options), so it generates an error saying that there's incorrect syntax near ‘;’.

Note If you have a composite join (a join based on multiple attributes), and you specify at least one expression but forget the others, neither syntax will trap the error. Similarly, other logical errors won't be trapped—for example, if you mistakenly type **ON C.OrderID = C.OrderID**.

OUTER

Outer joins are used to return matching rows from both tables based on some criterion, plus rows from the "preserved" table or tables for which there was no match.

You identify preserved tables with the **LEFT**, **RIGHT**, or **FULL** keywords. **LEFT** marks the left table as preserved, **RIGHT** marks the right table, and **FULL** marks both.

Outer joins apply all three query logical processing phases—namely, Cartesian product, **ON** filter, and adding outer rows. Outer rows added for rows from the preserved table with no match have **NULLs** for the attributes of the nonpreserved table.

The following query returns customers with their order IDs (just as would an inner join with the same **ON** clause), but it also returns a row for each customer with no orders because the keyword **LEFT** identifies the **Customers** table as preserved:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C
LEFT OUTER JOIN dbo.Orders AS O
ON C.CustomerID = O.CustomerID;
```

The keyword **OUTER** is optional because the mention of one of the keywords **LEFT**, **RIGHT**, or **FULL** implies an outer join. However, unlike inner joins, where most programmers typically don't specify the optional **INNER** keyword, most programmers (including me) typically do specify the **OUTER** keyword. I guess it *feels* more natural.

As I mentioned earlier, SQL Server 2005 will support the nonstandard proprietary syntax for outer joins only under a backward-compatibility flag. To enable the older syntax, change the Northwind database's compatibility mode to 80 (SQL Server 2000):

```
EXEC sp_dbcmpptlevel Northwind, 80;
```

Note Changing the compatibility mode of a database to an earlier version will prevent you from using the new language elements (for example, ranking functions, recursive queries, and so on). I'm just changing the compatibility mode to demonstrate the code. Once I'm done, I'll instruct you to turn it back to 90 (SQL Server 2005).

The old-style outer join was indicated in the **WHERE** clause, not the **FROM** clause. Instead of **=**, it used ***=** to represent a left outer join and **=*** to represent a right outer join. There was no support for a full outer join. For example, the following query returns customers with their order IDs, and customers with no orders:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE C.CustomerID *=O.CustomerID;
```

This syntax is very problematic because of the lack of separation between an **ON** filter and a **WHERE** filter. For example, if you want to return only customers with no orders, using ANSI syntax it's very simple:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C
LEFT OUTER JOIN dbo.Orders AS O
ON C.CustomerID = O.CustomerID
WHERE O.CustomerID IS NULL;
```

You get customers **FISSA** and **PARIS** back. The query initially applies the first three steps in query logical processing, yielding an intermediate virtual table containing customers with their orders (inner rows) and also customers with no orders (outer rows). For the outer rows, the attributes from the **Orders** table are **NULL**. The **WHERE** filter is subsequently applied to this intermediate result. Only the rows with a **NULL** in the join column from the nonpreserved side, which represent the customers with no orders, satisfy the condition in the **WHERE** clause.

If you attempt to write the query using the old-style syntax, you will get surprising results:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE C.CustomerID *= O.CustomerID
```



```
AND O.CustomerID IS NULL;
```

The query returns all 91 customers. Because there's no distinction between an ON clause and a WHERE clause, I specified both expressions in the WHERE clause separated by the logical operator AND. You have no control over which part of the filter will take place before adding the outer rows and which part will take place afterwards. That's at the sole discretion of SQL Server. By looking at the result, you can guess what SQL Server did. Logically, it applied the whole expression before adding outer rows. Obviously, there's no row in the Cartesian product for which **C.CustomerID = O.CustomerID** and **O.CustomerID IS NULL**. So the second phase in query logical processing yields an empty set. The third phase adds outer rows for rows from the preserved table (Customers) with no match. Because none of the rows matched the join condition, all customers are added back as outer rows. That's why this query returned all 91 customers.

Bearing in mind that you got this surprising result because both expressions were applied before adding the outer rows, and being familiar with query logical processing, there is a way to "fix" the problem. Remember that there's another filter available to you in a query—the HAVING filter. First write a join query without the filter to isolate outer rows. Even though the rows in the result of the join are unique, group them by all columns to allow the query to include a HAVING clause. Then put the filter that isolates the outer rows into the HAVING clause:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE C.CustomerID *= O.CustomerID
GROUP BY C.CustomerID, CompanyName, OrderID
HAVING OrderID IS NULL;
```

In this manner, you control which filter will be applied before adding outer rows and which will be applied after.

Important Keep in mind that I demonstrated the older proprietary syntax just to make you aware of its issues. It is of course strongly recommended that you refrain from using it and revise all code that does use it to the ANSI syntax. In short, don't try this at home!

When you're done experimenting with the old-style syntax, change the database's compatibility level back to 90 (SQL Server 2005):

```
EXEC sp_dbcmptlevel Northwind, 90;
```

In the [previous chapter](#), I provided a solution using subqueries for the minimum missing value problem. As a reminder, you begin with the table T1, which you create and populate by running the code in [Listing 5-1](#).

Listing 5-1: 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,
    datacol VARCHAR(10) NOT NULL
);
INSERT INTO dbo.T1(keycol, datacol) VALUES(1, 'e');
INSERT INTO dbo.T1(keycol, datacol) VALUES(2, 'f');
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');
```

Your task is to find the minimum missing key (in this case, 5) assuming the key starts at 1. I provided the following solution based on subqueries:

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

Remember that I provided a CASE expression that returns the value 1 if it is missing; otherwise, it returns the result of the

preceding query. You can solve the same problem—returning the minimum missing key when 1 exists in the table—by using the following outer join query between two instances of T1:

```
SELECT MIN(A.keycol + 1)
FROM dbo.T1 AS A
  LEFT OUTER JOIN dbo.T1 AS B
    ON B.keycol = A.keycol+1
WHERE B.keycol IS NULL;
```

The first step in the solution is applying the left outer join between two instances of T1, called A and B, based on the join condition *B.keycol = A.keycol + 1*. This step involves the first three query logical processing phases I described in Chapter 1 (Cartesian product, ON filter, and adding outer rows). For now, ignore the WHERE filter and the SELECT clause. The join condition matches each row in A with a row from B whose key value is 1 greater than A's key value. Because it's an outer join, rows from A that have no match in B are added as outer rows, producing the virtual table shown in [Table 5-2](#).

Table 5-2: Output of Step 1 in Minimum Missing Value Solution

A.keycol	A.datacol	B.keycol	B.datacol
1	e	2	f
2	f	3	a
3	a	4	b
4	b	NULL	NULL
6	c	7	d
7	d	NULL	NULL

Note that the outer rows represent the points before the gaps because the next highest key value is missing. The second step in the solution is to isolate only the points before the gaps, the WHERE clause filters only rows where *B.keycol* is NULL, producing the virtual table shown in [Table 5-3](#).

Table 5-3: Output of Step 2 in Minimum Missing Value Solution

A.keycol	A.datacol	B.keycol	B.datacol
4	b	NULL	NULL
7	d	NULL	NULL

Finally, the last step in the solution is to isolate the minimum *A.keycol* value, which is the minimum key value before a gap, and adds 1. The result is the requested minimum missing value.

The optimizer generates very similar plans for both queries, with identical costs. So you can use the solution that you feel more comfortable with. To me, the solution based on subqueries seems more intuitive.

Nonsupported Join Types

ANSI SQL supports a couple of join types that are not supported by T-SQL—natural join and union join. I haven't found practical applications for a union join, so I won't bother to describe or demonstrate it in this book.

A natural join is an inner join where the join condition is implicitly based on equating columns that share the same names in both tables. The syntax for a natural join, not surprisingly, is NATURAL JOIN. For example, the two queries shown next are logically equivalent, but only the second is recognized by SQL Server:

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C NATURAL JOIN dbo.Orders AS O;
```

and

```
SELECT C.CustomerID, CompanyName, OrderID
FROM dbo.Customers AS C
  JOIN dbo.Orders AS O
    ON O.CustomerID = O.CustomerID;
```

Further Examples of Joins

So far, I have demonstrated fundamental join types. There are other ways to categorize joins besides their fundamental type. In this section, I'll describe self joins, nonequijoins, queries with multiple joins, and semi joins.

Self Joins

A self join is simply a join between two instances of the same table. I've already shown examples of self joins without classifying them explicitly as such.

Here's a simple example of a self join between two instances of the Employees table, one representing employees (E), and the other representing managers (M):

```
USE Northwind;
```

```
SELECT E.FirstName, E.LastName AS emp,
       M.FirstName, M.LastName AS mgr
FROM   dbo.Employees AS E
       LEFT OUTER JOIN dbo.Employees AS M
         ON E.ReportsTo = M.EmployeeID;
```

The query produces the output shown in [Table 5-4](#), where the employees' names are returned along with their managers' names.

Table 5-4: Employees and Their Managers

emp	mgr
Nancy Davolio	Andrew Fuller
Andrew Fuller	NULL
Janet Leverling	Andrew Fuller
Margaret Peacock	Andrew Fuller
Steven Buchanan	Andrew Fuller
Michael Suyama	Steven Buchanan
Robert King	Steven Buchanan
Laura Callahan	Andrew Fuller
Anne Dodsworth	Steven Buchanan

I used a left outer join to include Andrew—the Vice President of Sales—in the result. He has a NULL in the ReportsTo column because he has no manager.

Note When joining two instances of the same table, you must alias at least one of the tables. This provides a unique name or alias to each instance so that there is no ambiguity in the result column names and in the column names in the intermediate virtual tables.

Nonequijoins

Equijoins are joins with a join condition based on an equality operator. Nonequijoins have operators other than equality in their join condition.

As an example, suppose that you need to generate all pairs of two different employees from an Employees table. Assume that currently the table contains employee IDs A, B, and C. A cross join would generate the following nine pairs:

```
A, A
A, B
A, C
B, A
B, B
B, C
```

C, A

C, B

C, C

Obviously, a "self" pair (x, x) that has the same employee ID twice is not a pair of two different employees. Also, for each pair (x, y), you will find its "mirror" pair (y, x) in the result. You need to return only one of the two. To take care of both issues, you can specify a join condition that filters pairs where the key from the left table is smaller than the key from the right table. Pairs where the same employee appears twice will be removed. Also, one of the mirror pairs (x, y) and (y, x) will be removed because only one will have a left key smaller than the right key.

The following query returns the required result, without mirror pairs and without self pairs:

```
SELECT E1.EmployeeID, E1.LastName, E1.FirstName,
       E2.EmployeeID, E2.LastName, E2.FirstName
FROM   dbo.Employees AS E1
       JOIN dbo.Employees AS E2
       ON E1.EmployeeID < E2.EmployeeID;
```

You can also calculate row numbers using a nonequijoin. As an example, the following query calculates row numbers for orders from the Orders table, based on increasing *OrderID*:

```
SELECT O1.OrderID, O1.CustomerID, O1.EmployeeID, COUNT(*)ASrn
FROM   dbo.Orders AS O1
       JOIN dbo.Orders AS O2
       ON O2.OrderID <= O1.OrderID
GROUP BY O1.OrderID, O1.CustomerID, O1.EmployeeID;
```

You can find similarities between this solution and the pre-SQL Server 2005 set-based solution I showed in the [previous chapter](#) using subqueries. The join condition here contains the same logical expression I used in a subquery before. After applying the first two phases in query logical processing (Cartesian product and ON filter), each order from O1 is matched with all orders from O2 that have a smaller or equal *OrderID*. This means that a row from O1 with a target row number *n* will be matched with *n* rows from O2. Each row from O1 will be duplicated in the result of the join *n* times. If this is confusing, bear with me as I try to demonstrate this logic with an example. Say you have orders with the following IDs (in order): x, y, and z. The result of the join would be the following:

x, x

y, x

y, y

z, x

z, y

z, z

The join created duplicates out of each row from O1—as many as the target row number. The next step is to collapse each group of rows back to one row, returning the count of rows as the row number:

x, 1

y, 2

z, 3

Note that you must include in the GROUP BY clause all attributes from O1 that you want to return. Remember that in an aggregate query, an attribute that you want to return in the SELECT list must appear in the GROUP BY clause. This query suffers from the same N^2 performance issues I described with the subquery solution. This query also demonstrates an "expand-collapse" technique, where the join achieves the expansion of the number of rows by generating duplicates, and the grouping achieves the collapsing of the rows allowing you to calculate aggregates.

I find the subquery technique more appealing because it's so much more intuitive. I find the "expand-collapse" technique to

be artificial and nonintuitive.

Remember that in both solutions to generating row numbers you used an aggregate function— a count of rows. Very similar logic can be used to calculate other aggregates either with a subquery or with a join (expand-collapse technique). I will elaborate on this technique in Chapter 6 in the "Running Aggregations" section. I'll also describe there scenarios in which I'd still consider using the "expand-collapse" technique even though I find it less intuitive than the subquery technique.

Multiple Joins

A query with multiple joins involves three or more tables. In this section, I'll describe both physical and logical aspects of multi-join queries.

Controlling the Physical Join Evaluation Order In a multi-join query with no outer joins, you can rearrange the order in which the tables are specified without affecting the result. The optimizer is aware of that and will determine the order in which it accesses the tables based on cost estimations. In the query's execution plan, you might find that the optimizer chose to access the tables in a different order than the one you specified in the query.

For example, the following query returns customer company name and supplier company name, where the supplier supplied products to the customer:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID

JOIN dbo.[OrderDetails] AS OD
    ON OD.OrderID = O.OrderID
JOIN dbo.Products AS P
    ON P.ProductID = OD.ProductID
JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID;
```

Examine the execution plan shown in [Figure 5-3](#), and you will find that the tables are accessed physically in a different order than the logical order specified in the query.

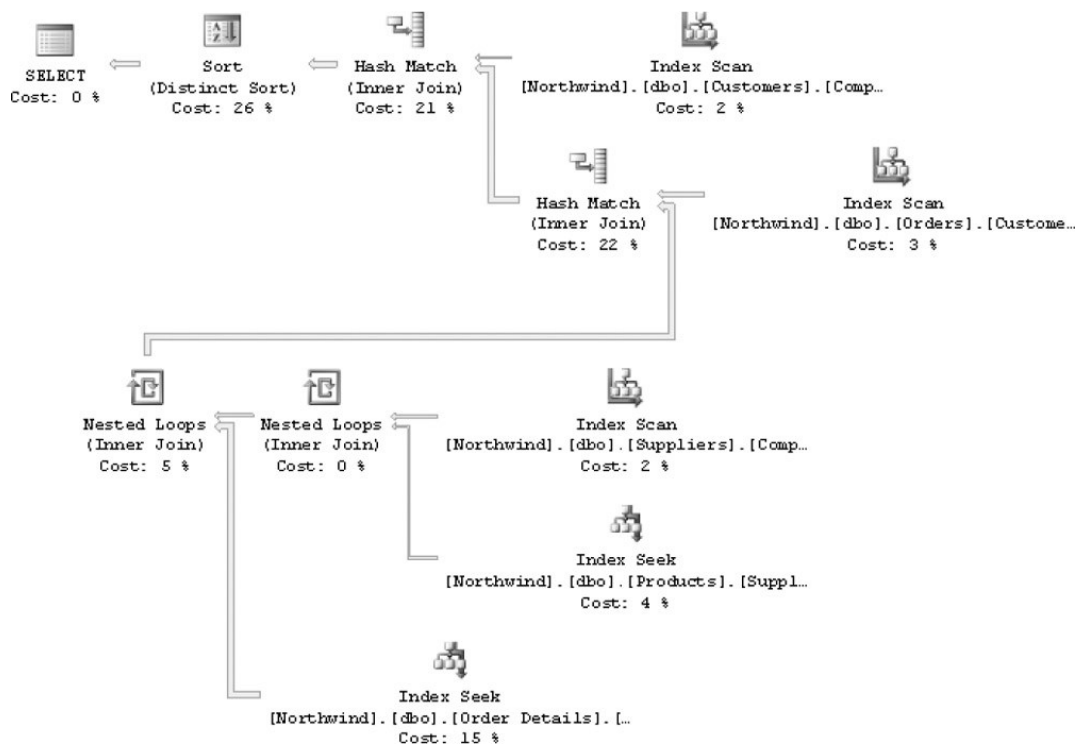


Figure 5-3: Execution plan for a multi-join query

If you suspect that a plan that accesses the tables in a different order than the one chosen by the optimizer will be more efficient, you can force the order of join processing by using one of two options. You can use the `FORCE ORDER` hint as

follows, forcing the optimizer to process the joins physically in the same order as the logical one:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
JOIN dbo.Orders AS O
  ON O.CustomerID = C.CustomerID
JOIN dbo.[OrderDetails] AS OD
  ON OD.OrderID = O.OrderID
JOIN dbo.Products AS P
  ON P.ProductID = OD.ProductID
JOIN dbo.Suppliers AS S
  ON S.SupplierID = P.SupplierID
OPTION (FORCE ORDER);
```

This query generates the execution plan shown in [Figure 5-4](#), where you can see that tables are accessed in the order they appear in the query.

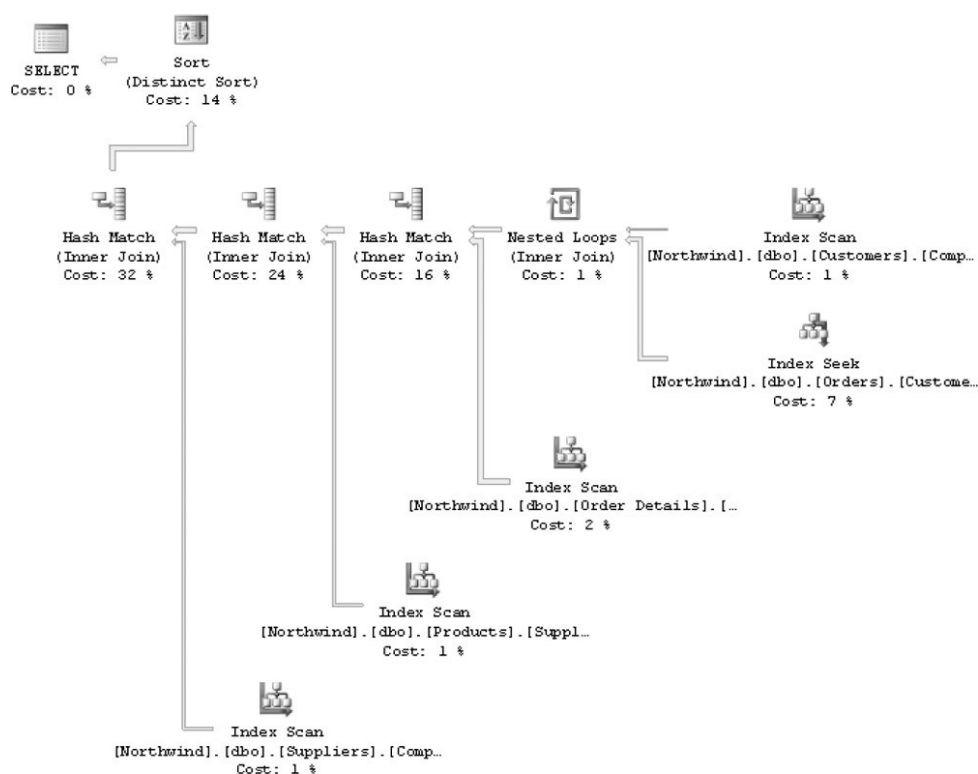


Figure 5-4: Execution plan for a multi-join query, forcing order

Another option to force order of join processing is to turn on the session option SET FORCEPLAN. This will affect all queries in the session.

Hints

Note that, in general, using hints to override the optimizer's choice of plan should be the last resort when dealing with performance issues. A hint is not a kind gesture; rather, you force the optimizer to use a particular route in optimization. If you introduce a hint in production code, that aspect of the plan becomes static (for example, the use of a particular index or a certain join algorithm). The optimizer will not make dynamic choices to accommodate changes in data volume and distribution.

There are several reasons that the optimizer might not produce an optimal plan and why a hint can improve performance.

First, the optimizer doesn't necessarily generate all possible execution plans for a query. If it did, the optimization phase could simply take too long. The optimizer calculates a threshold for optimization based on the input table sizes, and it will stop optimizing when that threshold is reached, yielding the plan with the lowest cost among the ones it did generate. This means that you won't necessarily get the optimal plan.

Second, optimization in many cases is based on data selectivity and density information, especially with regard to the choice of indexes and access methods. If statistics are not up to date or don't have a sufficient sampling rate, the optimizer might make inaccurate estimates.

Third, the key distribution histograms SQL Server maintains for indexed columns (and in some cases, nonindexed ones as well) have at most 200 steps. With many join conditions and filters, the difference between the selectivity/density information that the optimizer estimates and the actual information can be substantial in some cases, leading to inefficient plans.

Keep in mind though that you're not guaranteed to get the optimal plan. You should do everything in your power to avoid using hints in production code—for example, making sure that statistics are up to date, increasing the sampling rate if needed, and in some cases revising the query to help the optimizer make better choices. Use a hint only as a last resort if all other means fail. And if you do end up using a hint, revisit the code from time to time after doing more research or opening a support case with Microsoft.

Controlling the Logical Join Evaluation Order There are also cases where you might want to be able to control the logical order of join processing beyond the observable order in which the tables are specified in the FROM clause. For example, consider the previous request to return all pairs of customer company name and supplier company name, where the supplier supplied products to the customer. Suppose you were also asked to return customers that made no orders. By intuition, you'd probably make the following attempt, using a left outer join between Customers and Orders:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
    LEFT OUTER JOIN dbo.Orders AS O
        ON O.CustomerID = C.CustomerID
JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID
JOIN dbo.Products AS P
    ON P.ProductID = OD.ProductID
JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID;
```

The previous query returned 1,236 pairs of customer-supplier, and you expected this query to return 1,238 rows (because there are two customers that made no orders). However, this query returns the same result set as the previous one without the outer customers. Remember that the first join takes place only between the first two tables (Customers and Orders), applying the first three phases of query logical processing, and results in a virtual table. The resulting virtual table is then joined with the third table ([Order Details]), and so on.

The first join did in fact generate outer rows for customers with no orders, but the *OrderID* in those outer rows was NULL, of course. The second join—between the result virtual table and [Order Details]—removed those outer rows because an equijoin will never find a match based on a comparison to a NULL.

There are several ways to make sure that those outer customers will not disappear. One technique is to use a left outer join in all joins, even though logically you want inner joins between Orders, [Order Details], Products, and Suppliers:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
    LEFT OUTER JOIN dbo.Orders AS O
        ON O.CustomerID = C.CustomerID
    LEFT OUTER JOIN dbo.[OrderDetails] AS OD
        ON OD.OrderID = O.OrderID
    LEFT OUTER JOIN dbo.Products AS P
        ON P.ProductID = OD.ProductID
    LEFT OUTER JOIN dbo.Suppliers AS S
        ON S.SupplierID = P.SupplierID;
```

The left outer joins will keep the outer customers in the intermediate virtual tables. This query correctly produces 1,238 rows, including the two customers that made no orders. However, if you had orders with no related order details, order details with no related products, or products with no related suppliers, this query would have produced incorrect results. That is, you would have received outer rows for them that you didn't want to get.

Another option is to make sure the join with the Customers table is logically last. This can be achieved by using inner joins between all other tables, and finally a right outer join with Customers:

```
SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
```



```

FROM dbo.Orders AS O
  JOIN dbo.[Order Details] AS OD
    ON OD.OrderID = O.OrderID
  JOIN dbo.Products AS P
    ON P.ProductID = OD.ProductID
  JOIN dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID
  RIGHT OUTER JOIN dbo.Customers AS C
    ON O.CustomerID = C.CustomerID;

```

This scenario was fairly simple, but in cases where you mix different types of joins—not to mention new table operators that were added in SQL Server 2005 (for example, APPLY, PIVOT, UNPIVOT)—it might not be that simple. Furthermore, using left outer joins all along the way is very artificial. It's more intuitive to think of the query as a single left outer join, where the left table is the Customers table and the right table is the result of inner joins between all the other tables. Both ANSI SQL and T-SQL allow you to control the logical order of join processing:

```

SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN
    (dbo.Orders AS O
      JOIN dbo.[OrderDetails] AS OD
        ON OD.OrderID = O.OrderID

      JOIN dbo.Products AS P
        ON P.ProductID = OD.ProductID
      JOIN dbo.Suppliers AS S
        ON S.SupplierID = P.SupplierID)
    ON O.CustomerID = C.CustomerID;

```

Technically, the parentheses are ignored here, but I recommend you use them because they will help you write the query correctly. Using parentheses caused you to change another aspect of the query, which is the one that the language really uses to determine the logical order of processing. If you haven't guessed yet, it's the ON clause order. Specifying the ON clause *ON O.CustomerID = C.CustomerID* last causes the other joins to be logically processed first; the left outer join occurs logically between Customers and the inner join of the rest of the tables. You could write the query without parentheses, and it would mean the same thing:

```

SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN
    dbo.Orders AS O
      JOIN dbo.[Order Details] AS OD
        ON OD.OrderID = O.OrderID
      JOIN dbo.Products AS P
        ON P.ProductID = OD.ProductID
      JOIN dbo.Suppliers AS S
        ON S.SupplierID = P.SupplierID
    ON O.CustomerID = C.CustomerID;

```

Other variations that specify the ON clause that refers to *C.CustomerID* last include the following two:

```

SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    JOIN dbo.Products AS P
      JOIN dbo.[OrderDetails] AS OD
        ON P.ProductID = OD.ProductID
        ON OD.OrderID = O.OrderID
      JOIN dbo.Suppliers AS S
        ON S.SupplierID = P.SupplierID
    ON O.CustomerID = C.CustomerID;

SELECT DISTINCT C.CompanyName AS customer, S.CompanyName AS supplier
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    JOIN dbo.[OrderDetails] AS OD
      JOIN dbo.Products AS P
        ON S.SupplierID = P.SupplierID
        ON P.ProductID = OD.ProductID


```

```
ON OD.OrderID = O.OrderID
ON O.CustomerID = C.CustomerID;
```

There's an obvious disadvantage to not using parentheses—a decrease in the readability and clarity of code. Without parentheses, the queries are far from intuitive. But there's another issue, too.

Important You cannot play with the ON clause's order any way you'd like. There's a certain relationship that must be maintained between the order of the specified tables and the order of the specified ON clauses for the query to be valid. The relationship is called a *chiastic* relationship. A chiastic relationship is neither unique to SQL nor to computer science; rather, it appears in many fields, including poetry, linguistics, mathematics, and others. In an ordered series of items, this relationship correlates the first item with the last, the second with the next to last, and so on. For example, the word ABBA has a chiastic relationship between the letters. As an example for a chiastic relationship in mathematics, recall the arithmetic sequence I described in the last chapter: 1, 2, 3, ..., n. To calculate the sum of the elements, you make $n/2$ pairs based on a chiastic relationship ($1 + n$, $2 + n - 1$, $3 + n - 2$, and so on). The sum of each pair is always $1 + n$; *therefore, the total sum of the arithmetic sequence is $(1 + n) * n / 2 = (n + n^2) / 2$.*

Similarly, the relationship between the tables specified in the FROM clause and the ON clauses must be chiastic for the query to be valid. That is, the first ON clause can refer only to the immediate two tables right above it. The second ON clause can refer to the previously referenced tables and to an additional one right above them, and so on. [Figure 5-5](#) illustrates the chiastic relationship maintained in the last query. The code in the figure was slightly rearranged for readability.



```

dbo.Customers          AS C   LEFT OUTER JOIN
dbo.Orders             AS O           JOIN
dbo.[Order Details]    AS OD        JOIN
dbo.Products AS P JOIN
dbo.Suppliers AS S
    ON S.SupplierID = P.SupplierID
    ON P.ProductID = OD.ProductID
    ON OD.OrderID = O.OrderID
    ON O.CustomerID = C.CustomerID;
```

Figure 5-5: Chiastic relationship in a multi-join query

```
SELECT DISTINCT
    C.CompanyName AS customer,
    S.CompanyName AS supplier
FROM
```

Without using parentheses, the queries are not very readable and you need to be aware of the chiastic relationship in order to write a valid query. Conversely, if you do use parentheses, the queries are more readable and intuitive and you don't need to concern yourself with chiastic relationships, as parentheses force you to write correctly.

Semi Joins

Semi joins are joins that return rows from one table based on the existence of related rows in the other table. If you return attributes from the left table, the join is called a left semi join.

If you return attributes from the right table, it's called a right semi join.

There are several ways to achieve a semi join: using inner joins, subqueries, and set operations (which I'll demonstrate later in the chapter). Using an inner join, you select attributes from only one of the tables and apply DISTINCT. For example, the following query returns customers from Spain that made orders:

```
SELECT DISTINCT C.CustomerID, C.CompanyName
FROM dbo.Customers AS C
    JOIN dbo.Orders AS O
        ON O.CustomerID = C.CustomerID
WHERE Country = N'Spain';
```

You can also use the EXISTS predicate as follows:

```

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

```

If you're wondering whether there's any performance difference between the two, in this case the optimizer generates an identical plan for both. This plan is shown in [Figure 5-6](#).

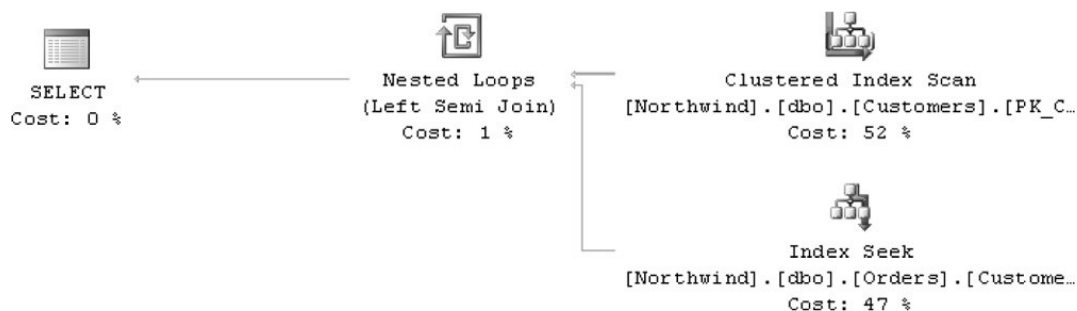


Figure 5-6: Execution plan for a left semi join

The inverse of a semi join is an anti-semi join, where you're looking for rows in one table based on their nonexistence in the other. You can achieve an anti-semi join (left or right) using an outer join, filtering only outer rows. For example, the following query returns customers from Spain that made no orders. The anti-semi join is achieved using an outer join:

```

SELECT C.CustomerID, C.CompanyName
FROM dbo.Customers AS C
LEFT OUTER JOIN dbo.Orders AS O
ON O.CustomerID = C.CustomerID
WHERE Country = N'Spain'
AND O.CustomerID IS NULL;

```

You can also use the NOT EXISTS predicate as follows:

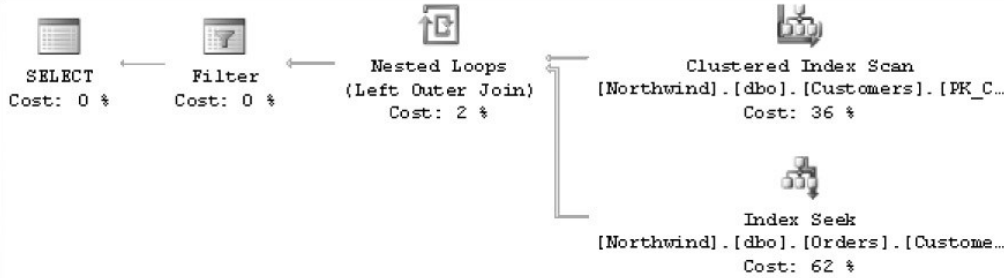
```

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

```

As you can see in the execution plans shown in [Figure 5-7](#) for the two query variations, the solution using the NOT EXISTS predicate performs better.

Query 1: Query cost (relative to the batch): 59%
SELECT C.CustomerID, C.CompanyName FROM dbo.Customers AS C LEFT OUTER JOIN



Query 2: Query cost (relative to the batch): 41%
SELECT CustomerID, CompanyName FROM dbo.Customers AS C WHERE Country = N'S

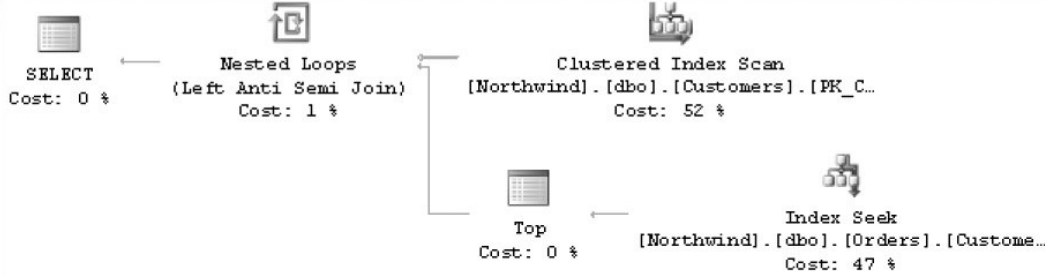


Figure 5-7: Execution plan for a left anti–semi join

The plan for the outer join solution shows that all orders for customers from Spain were actually processed. Let *c* equal the number of customers from Spain, and *o* equal the average number of orders per customer. You get *c* × *o* orders accessed. Then only the outer rows are filtered.

The plan for the NOT EXISTS solution is more efficient. Like the plan for the LEFT OUTER JOIN solution, this plan performs a seek within the index on *Orders.CustomerID* for each customer. However, the NOT EXISTS plan only checks whether or not a row with that customer ID was found (shown by the TOP operator)—while the plan for the outer join actually scans all index rows for each customer.

Sliding Total of Previous Year

The following exercise demonstrates a mix of several join categories: a self join, non-equijoin, and multi-join query. First create and populate the *MonthlyOrders* table by running the code in Listing 5-2.

Listing 5-2: Creating and populating the *MonthlyOrders* table

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

SELECT
    CAST(CONVERT(CHAR(6), OrderDate,112) + '01' AS DATETIME)
        AS ordermonth,
    COUNT(*) AS numorders
INTO dbo.MonthlyOrders
FROM dbo.Orders
GROUP BY CAST(CONVERT(CHAR(6), OrderDate,112) + '01' AS DATETIME);

CREATE UNIQUE CLUSTERED INDEX idx_ordermonth ON dbo.MonthlyOrders(ordermonth);
```

The table stores the total number of orders for each month as shown in Table 5-5.

Table 5-5: Contents of the *MonthlyOrders* Table

ordermonth	numorders
------------	-----------

1996-07-01 00:00:00.000	22
1996-08-01 00:00:00.000	25
1996-09-01 00:00:00.000	23
1996-10-01 00:00:00.000	26
1996-11-01 00:00:00.000	25
1996-12-01 00:00:00.000	31
1997-01-01 00:00:00.000	33
1997-02-01 00:00:00.000	29
1997-03-01 00:00:00.000	30
1997-04-01 00:00:00.000	31
1997-05-01 00:00:00.000	32
1997-06-01 00:00:00.000	30
1997-07-01 00:00:00.000	33
1997-08-01 00:00:00.000	33
1997-09-01 00:00:00.000	37
1997-10-01 00:00:00.000	38
1997-11-01 00:00:00.000	34
1997-12-01 00:00:00.000	48
1998-01-01 00:00:00.000	55
1998-02-01 00:00:00.000	54
1998-03-01 00:00:00.000	73
1998-04-01 00:00:00.000	74
1998-05-01 00:00:00.000	14

Notice that I used the DATETIME datatype for the *ordermonth* column. A valid date must include a day portion, so I just used the 1st of the month. When I'll need to present data, I'll get rid of the day portion. Storing the order month in a DATETIME datatype allows more flexible manipulations using DATETIME related functions.

The request is to return, for each month, a sliding total of the previous year. In other words, for each month *n*, return the total number of orders from month *n* minus 11 through month *n*.

The solution I will show is not the most efficient technique to achieve this task. I'm showing it here as an example of a mix between different join types and categories. Also, I'm assuming that there are no gaps in the sequence of months. In Chapter 6, you will find a focused discussion on running aggregates, including performance issues.

The following query returns the sliding total of the previous year for each month, generating the output shown in [Table 5-6](#):

Table 5-6: Sliding Total of Previous Year

frommonth	tomonth	numorders
199607	199706	337
199608	199707	348
199609	199708	356
199610	199709	370
199611	199710	382
199612	199711	391
199701	199712	408
199702	199801	430
199703	199802	455

199704	199803	498
199705	199804	541
199706	199805	523

```

SELECT
    CONVERT(CHAR(6), O1.ordermonth,112) AS frommonth,
    CONVERT(CHAR(6), O2.ordermonth,112) AS tomonth,
    SUM(O3.numorders) AS numorders
FROM dbo.Monthly Orders AS O1
    JOIN dbo.MonthlyOrders AS O2
        ON DATEADD(month, -11, O2.ordermonth) = O1.ordermonth
    JOIN dbo.Monthly Orders AS O3
        ON O3.ordermonth BETWEEN O1.ordermonth AND O2.ordermonth
GROUP BY O1.ordermonth, O2.ordermonth;

```

The query first joins two instances of `MonthlyOrders`, `O1` and `O2`. The two instances supply the boundary dates of the sliding year, `O1` for the lower boundaries (*frommonth*), and `O2` for the upper boundaries (*tomonth*). Therefore, the join condition is as follows: *ordermonth in O1 = ordermonth in O2 – 11 months*. For example, July 1996 in `O1` will match June 1997 in `O2`.

Once the boundaries are fixed, another join, to a third instance of `MonthlyOrders` (`O3`), matches to each boundary-pair row that falls within that range. In other words, each boundary-pair will find 12 matches, one for each month, assuming there were orders in all 12 months. The logic here is similar to the expand technique I was talking about earlier. Now that each boundary pair has been duplicated 12 times, once for each qualifying month from `O3`, you want to collapse the group back to a single row, returning the total number of orders for each group.

Note that this solution will return only pairs in which both boundaries exist in the data and are 11 months apart. It will not return high-bound months for which a low-bound month does not exist in the data. July 1996 is currently the earliest month that exists in the table. Therefore, June 1997 is the first high-bound month that appears in the result set. If you also want to get results with *tomonth* before June 1997, you need to change the first join to a right outer join. The right outer join will yield a NULL in the *frommonth* column for the outer rows. In order not to lose those outer rows in the second join, in the join condition you convert a NULL *frommonth* to the date January 1, 1900, ensuring that *frommonth* <= *tomonth*. Another option would be to use the minimum month within the 12-month range over which the total is calculated. For simplicity's sake, I'll use the former option.

Similarly, in the `SELECT` list you convert a NULL in the *frommonth* column to the minimum existing month.

To note that some ranges do not cover a whole year, return also the count of months involved in the aggregation.

Here's the complete solution, which returns the output shown in [Table 5-7](#):

Table 5-7: Sliding Total of Previous Year, Including All Months

frommonth	tomonth	numorders	nummonths
199607	199607	22	1
199607	199608	47	2
199607	199609	70	3
199607	199610	96	4
199607	199611	121	5
199607	199612	152	6
199607	199701	185	7
199607	199702	214	8
199607	199703	244	9
199607	199704	275	10
199607	199705	307	11
199607	199706	337	12
199608	199707	348	12

199609	199708	356	12
199610	199709	370	12
199611	199710	382	12
199612	199711	391	12
199701	199712	408	12
199702	199801	430	12
199703	199802	455	12
199704	199803	498	12
199705	199804	541	12
199706	199805	523	12

```

SELECT
    CONVERT(VARCHAR(6),
        COALESCE(O1.ordermonth,
            (SELECT MIN(ordermonth) FROM dbo.MonthlyOrders)),
        112) AS frommonth,
    CONVERT(VARCHAR(6), O2.ordermonth, 112) AS tomonth,
    SUM(O3.numorders) AS numorders,
    DATEDIFF(month,
        COALESCE(O1.ordermonth,
            (SELECT MIN(ordermonth) FROM dbo.MonthlyOrders)),
        O2.ordermonth) + 1 AS nummonths
FROM dbo.MonthlyOrders AS O1
RIGHT JOIN dbo.MonthlyOrders AS O2
    ON DATEADD(month, -11, O2.ordermonth) = O1.ordermonth
JOIN dbo.MonthlyOrders AS O3
    ON O3.ordermonth BETWEEN COALESCE(O1.ordermonth, '19000101')
        AND O2.ordermonth
GROUP BY O1.ordermonth, O2.ordermonth;

```

To clean up, drop the MonthlyOrders table:

```
DROP TABLE dbo.MonthlyOrders;
```

Join Algorithms

Join algorithms are the physical strategies SQL Server can use to process joins. Prior to SQL Server 7.0, only one join algorithm (called nested loops or loop join) was supported. Since version 7.0, SQL Server supports Merge and Hash join algorithms also.

Loop Join

A loop join scans one of the joined tables (the upper table in the graphical query plan), and for each row it searches the matching rows in the other joined table (the lower table in the plan).

Note The presence of a loop join operator in the execution plan does not indicate whether it's an efficient plan or not. A loop join is a default algorithm that can always be applied. The other algorithms have requirements—for example, the join must be an equijoin.

Using a loop join is efficient when there's an index on the join column in the larger table, allowing a seek followed by a partial scan. How efficient it is depends on where the index is positioned in this index optimization scale:

Worse Performance←no index (table scan per outer row)→nonclustered noncovering (seek + partial ordered scan + lookups)→clustered(seek + partial scan)→nonclustered covering (seek + partial scan)→nonclustered covering with included nonkey columns(seek + partial scan)→Best Performance.

The following query, which produces the plan shown in [Figure 5-8](#), is an example of a query for which the optimizer chooses the loop join operator:

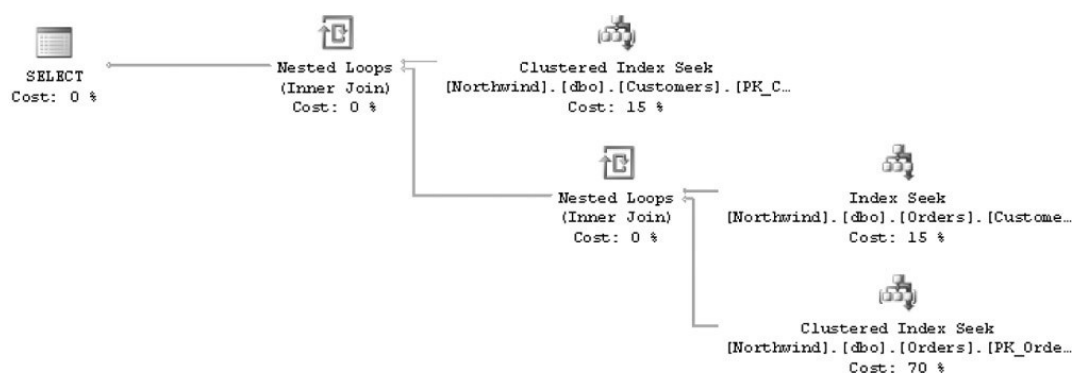


Figure 5-8: Execution plan for a loop join

```
SELECT C.CustomerID, C.CompanyName, O.OrderID, O.OrderDate
FROM dbo.Customers AS C
JOIN dbo.Orders AS O
    ON O.CustomerID = C.CustomerID
WHERE C.CustomerID = N'ALFKI';
```

The plan performs a seek within the clustered index on Customers to return the filtered customers. For each customer (in our case, there's only one), a loop join operator initiates a seek within the index on *Orders.CustomerID*. The seek is followed by a partial scan to get the row locators to all the customer's orders. For each row locator, another loop join operator (below and to the right of the first one) initiates a lookup of the data row. Note that this loop join operator does not correspond to a join in the query; rather, it is used to initiate the physical lookup operations. Try to think of it as an internal join between indexes on the Orders table. In the plan, the lookup shows up as a seek within the clustered index on Orders, because the table is clustered and the row locator is the clustering key.

Important With regard to joins and indexing, remember that joins are often based on foreign key/primary key relationships.

While an index (to enforce uniqueness) is automatically created when a primary key is declared, a foreign key declaration doesn't automatically create an index. Remember that for loop joins, typically an index on the join column in the larger table is preferable. So it's your responsibility to create that index explicitly.

Merge

A merge join is a very efficient join algorithm, which relies on two inputs that are sorted on the join columns. With a one-to-many join, a merge join operator scans each input only once—hence, the superiority over the other operators. To have two sorted inputs, the optimizer can use clustered, or better yet, nonclustered covering indexes with the first column or columns being the join column(s). SQL Server will start scanning both sides, moving forward first through the "many" side (say, "one" side is *x* and "many" is *x, x, x, x, y*). SQL Server will step through the "one" side when the join column value on the "many" side changes (the "one" side steps through *y*, and then the "many" side steps through *y, y, y, z*). Ultimately, each side is scanned only once, in an ordered fashion. Things become more complicated when you have a many-to-many join, where the optimizer might still use a merge join operator with rewind logic.

When you see a merge join in the plan, it's usually a good sign.

As an example, the following query joins Orders and [Order Details] on equal *OrderID* values:

```
SELECT O.OrderID, O.OrderDate, OD.ProductID, OD.Quantity
FROM dbo.Orders AS O
JOIN dbo.[Order Details] AS OD
    ON O.OrderID = OD.OrderID;
```

Both tables have a clustered index on *OrderID*, so the optimizer chooses a merge join. The merge join operator appears in the plan for this query, shown in [Figure 5-9](#).

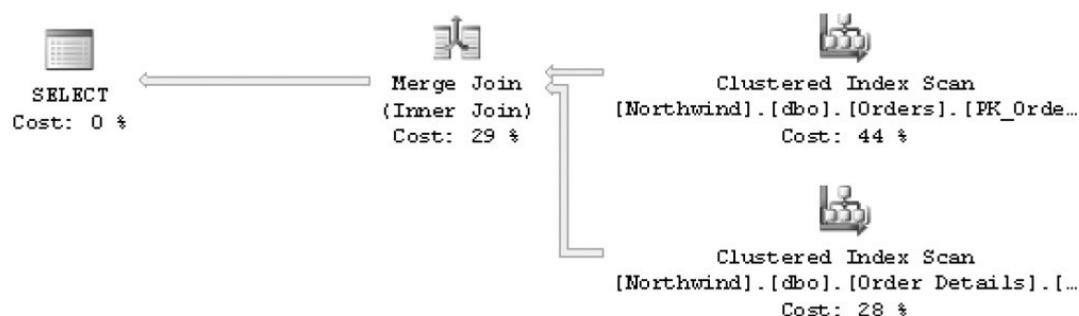


Figure 5-9: Execution plan for a merge join

In some cases, the optimizer might decide to use a merge join even when one of the inputs is not presorted by an index, especially if that input is fairly small. In such a case, you will see that input scanned and then sorted, as in the execution plan of the following query. [Figure 5-10](#) shows the execution plan for this query:

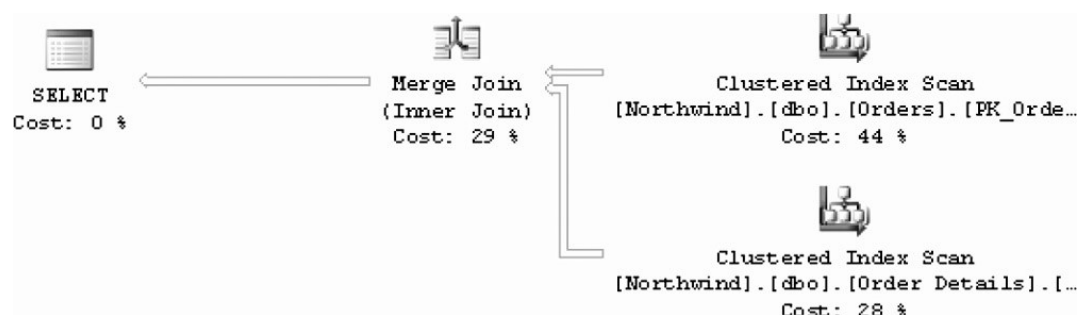


Figure 5-10: Execution plan for a merge join with sort

```
SELECT C.CustomerID, CompanyName, ContactName, ContactTitle,
       OrderID, OrderDate
FROM   dbo.Customers AS C
JOIN   dbo.Orders AS O
       ON O.CustomerID = C.CustomerID;
```

Hash

The hash join operator is typically chosen by the optimizer when good indexes on the join columns are missing. If you don't create appropriate indexes for the join, the optimizer creates a hash table as an alternative searching structure to balanced trees. Balanced trees are usually more efficient to use as a searching structure than hash tables, but they are more expensive to create. Occasionally, you see execution plans where the optimizer decides that it's worthwhile to create a temporary index (an Index Spool operator). But usually, for ad-hoc queries, it's more expensive to create a temporary index (balanced tree), use it, and drop it than to create a hash table and use it.

The optimizer uses the smaller input of the two as the build input for the hash table. It distributes the rows (relevant attributes for query) from the build input into buckets, based on a hash function applied to the join column values. The hash function is chosen to create a predetermined number of buckets of fairly equal size. Say you have a garage with a large number of tools and items. If you don't organize them in a particular manner, every time you look for an item you need to scan all of them. This is similar to a table scan. Of course, you will want to organize the items in groups and shelves by some criteria—for example, by functionality, size, color, and so on. You'd probably choose a criterion that would result in fairly equal sized, manageable groups.

The criterion you would use is analogous to the hash function, and a shelf or group of items is analogous to the hash bucket. Once the items in the garage are organized, every time you need to look for one, you apply to it the same criterion as the one you used to organize the items, go directly to the relevant shelf, and scan it.

Tip When you see a hash join operator in the execution plan, it should be a warning to you that your data might be missing an important index. Of course, if you're running an ad-hoc query that is not repeated frequently, you might be happy with the hash join. However, if it's a production query that is invoked frequently, you might want to create the missing indexes.

To demonstrate a hash join, first create copies of the tables `Orders` and `Order Details` without indexes on the join columns:

```
SELECT * INTO dbo.MyOrders FROM dbo.Orders;
SELECT * INTO dbo.MyOrderDetails FROM dbo.[Order Details];
```

Next, run the following query:

```
SELECT O.OrderID, O.OrderDate, OD.ProductID, OD.Quantity
FROM dbo.MyOrders AS O
JOIN dbo.MyOrderDetails AS OD
ON O.OrderID = OD.OrderID;
```

You will see the Hash Match operator in the execution plan generated for the query as shown in [Figure 5-11](#):

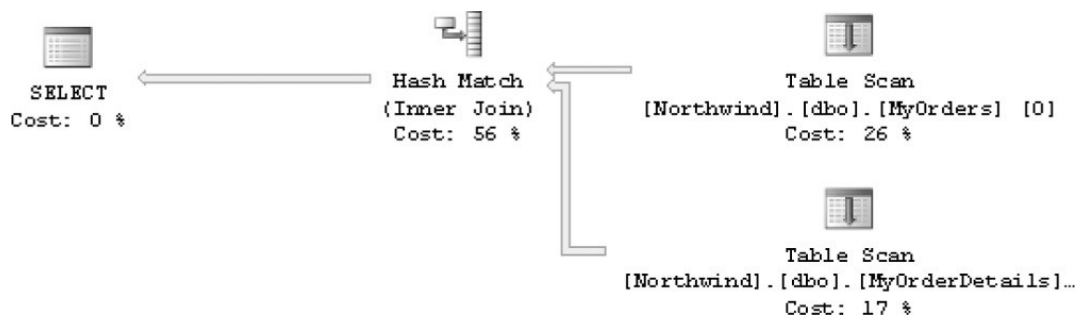


Figure 5-11: Execution plan for a hash join

When you're done, drop the tables you just created:

```
DROP TABLE dbo.MyOrders;
DROP TABLE dbo.MyOrderDetails;
```

Forcing a Join Strategy

You can force the optimizer to use a particular join algorithm, provided that it's technically supported for the given query. You do so by specifying a hint between the keyword or keywords representing the join type (for example, INNER, LEFT OUTER) and the JOIN keyword. For example, the following query forces a loop join:

```
SELECT C.CustomerID, C.CompanyName, O.OrderID
FROM dbo.Customers AS C
INNER LOOP JOIN dbo.Orders AS O
ON O.CustomerID = C.CustomerID;
```

Note With inner joins, when forcing a join algorithm, the keyword INNER is not optional. With outer joins, the OUTER keyword is still optional. For example, you can use LEFT LOOP JOIN or LEFT OUTER LOOP JOIN.

Using the older join syntax, you can't specify a different join algorithm for each join; rather, one algorithm will be used for all joins in the query. You do so by using the OPTION clause, like so:

```
SELECT C.CustomerID, C.CompanyName, O.OrderID
FROM dbo.Customers AS C, dbo.Orders AS O
WHERE O.CustomerID = C.CustomerID
OPTION (LOOPJOIN);
```

Tip Keep in mind the discussion held earlier in the chapter regarding using hints to override the optimizer's choices. Limit their use, and try to exhaust all other means before you introduce such a hint in production code.

Separating Elements

At this point, you have a chance to put your knowledge of joins and the key techniques you learned so far into action. Here I'll present a generic form of a fairly tough problem that has many practical applications in production. Create and populate a table called Arrays by running the code in [Listing 5-3](#).

Listing 5-3: Creating and populating the table Arrays

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

```

CREATE TABLE dbo.Arrays
(
    arrid VARCHAR(10)    NOT NULL PRIMARY KEY,
    array VARCHAR(8000)  NOT NULL
)

INSERT INTO Arrays(arrid, array) VALUES('A', '20,22,25,25,14');
INSERT INTO Arrays(arrid, array) VALUES('B', '30,33,28');
INSERT INTO Arrays(arrid, array) VALUES('C', '12,10,8,12,12,13,12,14,10,9');
INSERT INTO Arrays(arrid, array) VALUES('D', '-4,-6,-4,-2');

```

The table contains arrays of elements separated by commas. Your task is to write a query that generates the result shown in [Table 5-8](#).

Table 5-8: Arrays Split to Elements

arrid	pos	val
A	1	20
A	2	22
A	3	25
A	4	25
A	5	14
B	1	30
B	2	33
B	3	28
C	1	12
C	2	10
C	3	8
C	4	12
C	5	12
C	6	13
C	7	12
C	8	14
C	9	10
D	1	-4
D	2	-6
D	3	-4
D	4	-2

The request is to normalize data that is not in first normal form—no multivalued attributes. The result set should have a row for each array element, including the array ID, the element's position within the array, and the element value. The solution is presented in the following paragraphs.

Before you even start coding, it's always a good idea to identify the steps in the solution and resolve them logically. It's often a good starting point to think in terms of the number of rows in the target and consider how that is related to the number of rows in the source. Obviously, here you need to generate multiple rows in the result from each row in Arrays. In other words, as the first step, you need to generate duplicates.

You already know that to generate duplicates, you can join the Arrays table with an auxiliary table of numbers. Here the join is not a simple cross join and a filter on a fixed number of rows. The number of duplicates here should equal the number of elements in the array. Each element is identified by a preceding comma (except for the first element, which we must not forget). So the join condition can be based on the existence of a comma in the n th character position in the array,

where n comes from the Nums table.

Obviously, you wouldn't want to check characters beyond the length of the array, so you can limit n to the array's length. The following query implements the first step of the solution, generating the output shown in [Table 5-9](#):

Table 5-9: Solution to Separating Elements Problem, Step 1

arrid	array	n
A	20,22,25,25,14	3
A	20,22,25,25,14	6
A	20,22,25,25,14	9
A	20,22,25,25,14	12
B	30,33,28	3
B	30,33,28	6
C	12,10,8,12,12,13,12,14,10,9	3
C	12,10,8,12,12,13,12,14,10,9	6
C	12,10,8,12,12,13,12,14,10,9	8
C	12,10,8,12,12,13,12,14,10,9	11
C	12,10,8,12,12,13,12,14,10,9	14
C	12,10,8,12,12,13,12,14,10,9	17
C	12,10,8,12,12,13,12,14,10,9	20
C	12,10,8,12,12,13,12,14,10,9	23
C	12,10,8,12,12,13,12,14,10,9	26
D	-4,-6,-4,-2	3
D	-4,-6,-4,-2	6
D	-4,-6,-4,-2	9

```
SELECT arrid, array, n
FROM dbo.Arrays
JOIN dbo.Nums
  ON n <= LEN(array)
  AND SUBSTRING(array, n, 1) = ',';
```

You have almost generated the correct number of duplicates for each array, along with the n value representing the matching comma's position. You have one fewer duplicate than the desired number of duplicates for each array. For example, array A has five elements but you have only four rows. The reason that a row is missing for each array is that there's no comma preceding the first element in the array. To fix this small problem, concatenate a comma and the array to generate the first input of the SUBSTRING function:

```
SELECT arrid, array, n
FROM dbo.Arrays
JOIN dbo.Nums
  ON n >= LEN(array)
  AND SUBSTRING(',' + array, n, 1) = ',';
```

As you can see in the output shown in [Table 5-10](#), each array now produces an additional row in the result with $n = 1$.

Table 5-10: Solution to Separating Elements Problem, Step 2

arrid	array	N
A	20,22,25,25,14	1
A	20,22,25,25,14	4
A	20,22,25,25,14	7
A	20,22,25,25,14	10

A	20,22,25,25,14	13
B	30,33,28	1
B	30,33,28	4
B	30,33,28	7
C	12,10,8,12,12,13,12,14,10,9	1
C	12,10,8,12,12,13,12,14,10,9	4
C	12,10,8,12,12,13,12,14,10,9	7
C	12,10,8,12,12,13,12,14,10,9	9
C	12,10,8,12,12,13,12,14,10,9	12
C	12,10,8,12,12,13,12,14,10,9	15
C	12,10,8,12,12,13,12,14,10,9	18
C	12,10,8,12,12,13,12,14,10,9	21
C	12,10,8,12,12,13,12,14,10,9	24
C	12,10,8,12,12,13,12,14,10,9	27
D	-4,-6,-4,-2	1
D	-4,-6,-4,-2	4
D	-4,-6,-4,-2	7
D	-4,-6,-4,-2	10

Also, because all characters in `' + array` are one character to the right than in the original array, all n values are greater than before by one. That's actually even better for us because now n represents the starting position of the corresponding element within the array.

The third step is to extract from each row the element starting at the n th character. You know where the element starts—at the n th character—but you need to figure out its length. The length of the element is the position of the next comma minus the element's starting position (n). You use the `CHARINDEX` function to find the position of the next comma. You will need to provide the function with the n value as the third argument to tell it to start looking for the comma at or after the n th character, and not from the beginning of the string. Just keep in mind that you'll face a very similar problem here to the one that caused you to get one less duplicate than the number of elements. Here, there's no comma after the last element. Just as you added a comma before the first element earlier, you can now add one at the end. The following query shows the third step in the solution and generates the output shown in [Table 5-11](#):

Table 5-11:
Solution to
Separating
Elements
Problem, Step 3

arrid	element
A	20
A	22
A	25
A	25
A	14
B	30
B	33
B	28
C	12
C	10
C	8

C	12
C	12
C	13
C	12
C	14
C	10
C	9
D	-4
D	-6
D	-4
D	-2

```
SELECT arrid,
       SUBSTRING(array, n, CHARINDEX(',', array + ',', n) - n) AS element
FROM dbo.Arrays
     JOIN dbo.Nums
       ON n <= LEN(array)
      AND SUBSTRING(',', array, n, 1) = ',';
```

Note that the element result column is currently a character string. You might want to convert it to a more appropriate datatype (for example, an integer in this case).

Finally, the last step in the solution is to calculate the position of each element within the array. This is a tricky step.

You first need to figure out what determines the position of an element within an array. The position is the number of commas in the original array up to the *n*th character, plus one. Once you figure this out, you need to come up with an expression that will calculate this. You want to avoid writing a T-SQL user-defined function, as it will slow the query down. If you come up with an inline expression that uses only built-in functions, you will get a very fast solution. To phrase the problem more technically, you need to take the first *n* characters (*LEFT(array, n)*) and count the number of commas within that substring. The problem is that most string functions have no notion of repetitions or multiple occurrences of a substring within a string. There is one built-in function, though, that does—*REPLACE*. This function replaces each occurrence of a certain substring (call it *oldsubstr*) within a string (call it *str*) with another substring (call it *newsubstr*). You invoke the function with the aforementioned arguments in the following order: *REPLACE(str, oldsubstr, newsubstr)*. Here's an interesting way we can use the *REPLACE* function: *REPLACE(LEFT(array, n), ',', '')*. Here *str* is the first *n* characters within the array (*LEFT(array, n)*), *oldsubstr* is a comma, and *newsubstr* is an empty string. We replace each occurrence of a comma within the substring with an empty string. Now, what can you say about the difference in length between the original substring (*n*) and the new one? The new one will obviously be *n - num_commas*, where *num_commas* is the number of commas in *str*. In other words, *n - (n - num_commas)* will give you the number of commas. Add one, and you have the position of the element within the array. Use the *LEN* function to return the number of characters in *str* after removing the commas. Here's the complete expression that calculates *pos*:

```
n - LEN(REPLACE(LEFT(array, n), ',', '')) + 1 AS pos
```

Using the *REPLACE* function to count occurrences of a string within a string is a trick that can come in handy.

Tip In SQL Server 2005, you can use the *ROW_NUMBER()* function to calculate *pos*:

```
ROW_NUMBER() OVER(PARTITION BY arrid ORDER BY n) AS pos
```

The following query shows the final solution to the problem, including the position calculation:

```
SELECT arrid,
       n - LEN(REPLACE(LEFT(array, n), ',', '')) + 1 AS pos,
       CAST(SUBSTRING(array, n, CHARINDEX(',', array + ',', n) - n)
           AS INT) AS element
FROM dbo.Arrays
     JOIN dbo.Nums
       ON n >= LEN(array)
      AND SUBSTRING(',', array, n, 1) = ',';
```

In SQL Server 2005, you can use a recursive CTE to separate elements without the need to use an auxiliary table of

numbers.

```
WITH SplitCTE AS
(
    SELECT arrid, 1 AS pos, 1 AS startpos,
           CHARINDEX(' ', array + ',') - 1 AS endpos
    FROM dbo.Arrays
    WHERE LEN(array) > 0

    UNION ALL

    SELECT Prv.arrid, Prv.pos + 1, Prv.endpos + 2,
           CHARINDEX(' ', Cur.array + ', ', Prv.endpos + 2) - 1
    FROM SplitCTE AS Prv
    JOIN dbo.Arrays AS Cur
        ON Cur.arrid = Prv.arrid
        AND CHARINDEX(' ', Cur.array + ', ', Prv.endpos + 2) > 0
)

SELECT A.arrid, pos,
       CAST(SUBSTRING(array, startpos, endpos-startpos+1) AS INT) AS element
FROM dbo.Arrays AS A
JOIN SplitCTE AS S
    ON S.arrid = A.arrid
ORDER BY arrid, pos;
```

The CTE calculates the start and end position of each element. The anchor member calculates the values for the first element within each array. The recursive member calculates the values of the "next" elements, terminating when no "next" elements are found. The *pos* column is initialized with the constant 1, and incremented by 1 in each iteration. The outer query joins the Arrays table with the CTE, and it extracts the individual elements of the arrays based on the start and end positions calculated by the CTE. This solution is a bit slower than the previous one, but it has the advantage of not requiring an auxiliary table of numbers.

Once I posted this puzzle in a private SQL trainer's forum. One of the trainers posted a very witty solution that one of his colleagues came up with. Here it is:

```
SELECTC AST(arrid AS VARCHAR(10)) AS arrid,
       REPLACE(array, ' ', CHAR(10)+CHAR(13)
       + CAST(arrid AS VARCHAR(10)) + SPACE(10))AS value
FROM dbo.Arrays;
```

First examine the solution to see whether you can figure it out, and then run it with Results To Text output mode. You will get the output shown in [Table 5-12](#), which "seems" correct.

Table 5-12:
Output of
Solution to
Separating
Elements
Problem that
"Seems"
Correct

arrid	value
A	20
A	22
A	25
A	25
A	14
B	30
B	33
B	28
C	12

C	10
C	8
C	12
C	12
C	13
C	12
C	14
C	10
C	9
D	-4
D	-6
D	-4
D	-2

This solution replaces each comma with a newline (`CHAR(10)+CHAR(13)) + array id + 10` spaces. It seems correct when you run it in text mode, but it isn't. If you run it in grid output mode, you will see that the output really contains only one row for each array.

Set Operations

You can think of joins as *horizontal* operations between tables, generating a virtual table that contains columns from both tables. This section covers *vertical* operations between tables, including UNION, EXCEPT, and INTERSECT. Any mention of *set operations* in this section refers to these vertical operations.

A set operation accepts two tables as inputs, each resulting from a query specification. For simplicity's sake, I'll just use the term *inputs* in this section to describe the input tables of the set operations.

UNION returns the unified set of rows from both inputs, EXCEPT returns the rows that appear in the first input but not the second, and INTERSECT returns rows that are common to both inputs.

ANSI SQL:1999 defines native operators for all three set operations, each with two nuances: one optionally followed by DISTINCT (the default) and one followed by ALL. SQL Server 2000 supported only two of these set operators, UNION and UNION ALL. SQL Server 2005 added native support for the set operators EXCEPT and INTERSECT. Currently, SQL Server does not support the optional use of DISTINCT for set operations. This is not a functional limitation because DISTINCT is implied when you don't specify ALL. I will discuss solutions to all set operations, with both nuances, in both versions.

Like joins, these set operations always operate on only two inputs, generating a virtual table as the result. You might feel comfortable calling the input tables *left* and *right* as with joins, or you might feel more comfortable referring to them as the *first* and *second* input tables.

Before I describe each set operation in detail, let's get a few technicalities regarding how set operations work out of the way.

The two inputs must have the same number of columns, and corresponding columns must have the same datatype, or at least be implicitly convertible. The column names of the result are determined by the first input.

An ORDER BY clause is not allowed in the individual table expressions. All other logical processing phases (joins, filtering, grouping, and so on) are supported on the individual queries except the TOP option.

Conversely, ORDER BY is the only logical processing phase supported directly on the final result of a set operation. If you specify an ORDER BY clause at the end of the query, it will be applied to the final result set. None of the other logical processing phases are allowed directly on the result of a set operation. I will provide alternatives later in the chapter.

Set operations work on complete rows from the two input tables. Note that when comparing rows between the inputs, set operations treat NULLs as equal, just like identical known values. In this regard, set operations are not like query filters (ON, WHERE, HAVING), which as you recall do not treat NULLs as equal.

UNION

UNION generates a result set combining the rows from both inputs. The following sections describe the differences between UNION (implicit DISTINCT) and UNION ALL.

UNION DISTINCT

Specifying UNION without the ALL option combines the rows from both inputs and applies a DISTINCT on top (in other words, removes duplicate rows).

For example, the following query returns all occurrences of *Country*, *Region*, *City* that appear in either the Employees table or the Customers table, with duplicate rows removed:

```
USE Northwind;
```

```
SELECT Country, Region, City FROM dbo.Employees
UNION
SELECT Country, Region, City FROM dbo.Customers;
```

The query returns 71 unique rows.

UNION ALL

You can think of UNION ALL as UNION without duplicate removal. That is, you get one result set containing all rows from both inputs, including duplicates. For example, the following query returns all occurrences of Customer, Region, City from both tables:

```
SELECT Country, Region, City FROM dbo.Employees
UNION ALL
SELECT Country, Region, City FROM dbo.Customers;
```

Because the Employees table has 9 rows and the Customers table has 91 rows, you get a result set with 100 rows.

EXCEPT

EXCEPT allows you to identify rows that appear in the first input but not in the second.

EXCEPT DISTINCT

EXCEPT DISTINCT returns distinct rows that appear in the first input but not in the second input. To achieve EXCEPT, programmers usually use the NOT EXISTS predicate, or an outer join filtering only outer rows, as I demonstrated earlier in the "[Semi Joins](#)" section. However, those solutions treat two NULLs as different from each other. For example, (UK, NULL, London) will not be considered equal to (UK, NULL, London). If both tables contain such a row, input1 EXCEPT input2 is not supposed to return it, yet the NOT EXISTS and outer join solutions will.

SQL Server versions prior to 2005 did not have support for the EXCEPT operator. The following code has a solution that is compatible with versions of SQL Server prior to SQL Server 2005 and that is logically equivalent to EXCEPT DISTINCT:

```
SELECT Country, Region, City
FROM (SELECT DISTINCT 'E' AS Source, Country, Region, City
      FROM dbo.Employees
      UNION ALL
      SELECT DISTINCT 'C', Country, Region, City
      FROM dbo.Customers) AS UA
GROUP BY Country, Region, City
HAVING COUNT(*) = 1 AND MAX(Source) = 'E';
```

The derived table query unifies the distinct rows from both inputs, assigning an identifier of the source input to each row (a result column called source). Rows from Employees are assigned the identifier E, and rows from Customers are assigned C. The outer query groups the rows by *Country*, *Region*, *City*. The HAVING clause keeps only groups that have one row (meaning that the row appeared only in one of the inputs) and a maximum source identifier of E (meaning that the row was from Employees). Logically, that's EXCEPT DISTINCT.

In SQL Server 2005, things are a bit simpler:

```
SELECT Country, Region, City FROM dbo.Employees
EXCEPT
```

```
SELECT Country, Region, City FROM dbo.Customers;
```

Note that of the three set operations, only EXCEPT is asymmetrical. That is, `input1 EXCEPT input2` is not the same as `input2 EXCEPT input1`.

For example, the query just shown returned the two cities that appear in Employees but not in Customers. The following query returns 66 cities that appear in Customers but not in Employees:

```
SELECT Country, Region, City FROM dbo.Customers
EXCEPT
SELECT Country, Region, City FROM dbo.Employees;
```

EXCEPT ALL

EXCEPT ALL is trickier than EXCEPT DISTINCT and has not yet been implemented in SQL Server. Besides caring about the existence of a row, it also cares about the number of occurrences of each row. Say you request the result of `input1 EXCEPT ALL input2`. If a row appears n times in *input1* and m times in *input2* (both n and m can be ≥ 0), it will appear $\text{MAX}(0, n - m)$ times in the output. That is, if n is greater than m , the row will appear $n - m$ times in the result; otherwise, it won't appear in the result at all.

The following query demonstrates how you can achieve EXCEPT ALL using a solution compatible with versions of SQL Server prior to SQL Server 2005:

```
SELECT Country, Region, City
FROM (SELECT Country, Region, City,
      MAX(CASE WHEN Source = 'E' THEN Cnt ELSE 0 END)ECnt,
      MAX(CASE WHEN Source = 'C' THEN Cnt ELSE 0 END)CCnt
FROM (SELECT 'E' AS Source,
      Country, Region, City, COUNT(*) AS Cnt
FROM dbo.Employees
GROUP BY Country, Region, City

      UNION ALL

      SELECT 'C', Country, Region, City, COUNT(*)
FROM dbo.Customers
GROUP BY Country, Region, City)AS UA
GROUP BY Country, Region, City) AS P
JOIN dbo.Nums
ON n <= ECnt - CCnt;
```

The derived table UA has a row for each distinct source row from each input, along with the source identifier (E for Employees, C for Customers) and the number of times (*Cnt*) it appears in the source.

The query generating the derived table P groups the rows from UA by *Country*, *Region*, and *City*. It uses a couple of MAX (CASE...) expressions to return the counts of duplicates from both sources in the same result row, calling them *ECnt* and *CCnt*. This is a technique to pivot data, and I'll talk about it in detail in Chapter 6. At this point, each distinct occurrence of *Country*, *Region*, *City* has a single row in P, along with the count of duplicates it had in each input. Finally, the outer query joins P with Nums to generate duplicates. The join condition is $n \leq ECnt - CCnt$. If you think about it, you will get the exact number of duplicates dictated by EXCEPT ALL. That is, if $ECnt - CCnt$ is greater than 0, you will get that many duplicates; otherwise, you'll get none.

Even though you don't have a native operator for EXCEPT ALL in SQL Server 2005, you can easily generate the logical equivalent using EXCEPT and the ROW_NUMBER function. Here's the solution:

```
WITH EXCEPT_ALL
AS
(
  SELECT
    ROW_NUMBER()
      OVER(PARTITION BY Country, Region, City
           ORDER BY Country, Region, City) AS rn,
    Country, Region, City
  FROM dbo.Employees

  EXCEPT
```

```

SELECT
    ROW_NUMBER( )
        OVER(PARTITION BY Country, Region, City
              ORDER BY Country, Region, City) AS rn,
    Country, Region, City
FROM dbo.Customers
)
SELECT Country, Region, City
FROM EXCEPT_ALL;

```

To understand the solution, I suggest that you first highlight sections (queries) within it and run them separately. This will allow you to examine the intermediate result sets and get a better idea of what the following paragraph tries to explain.

The code first assigns row numbers to the rows of each of the inputs, partitioned by the whole attribute list. The row numbers will number the duplicate rows within the input. For example, a row that appears five times in Employees and three times in Customers will get row numbers 1 through 5 in the first input, and row numbers 1 through 3 in the second input. You then apply `input1 EXCEPT input2`, and get rows (including the *rn* attribute) that appear in *input1* but not in *input2*. If row R appears 5 times in *input1* and 3 times in *input2*, you get the following result:

```
{(R, 1), (R, 2), (R, 3), (R, 4), (R, 5)}
```

EXCEPT

```
{(R, 1), (R, 2), (R, 3)}
```

And this produces the following result:

```
{(R, 4), (R, 5)}
```

In other words, R will appear in the result exactly the number of times mandated by EXCEPT ALL. I encapsulated this logic in a CTE to return only the original attribute list without the row number, which is what EXCEPT ALL would do.

INTERSECT

INTERSECT returns rows that appear in both inputs.

To achieve INTERSECT, programmers usually use the EXISTS predicate or an inner join, as I demonstrated earlier in the "Semi Joins" section. However, as I explained earlier, those solutions treat two NULLs as different from each other, and set operations are supposed to treat them as equal.

Support for the INTERSECT operator was introduced in SQL Server 2005, but only in the variation with the implicit DISTINCT.

INTERSECT DISTINCT

To achieve the logical equivalent of INTERSECT DISTINCT pre-SQL Server 2005 versions, you can use the following solution:

```

SELECT Country, Region, City
FROM (SELECT DISTINCT Country, Region, City FROM dbo.Employees
      UNION ALL
      SELECT DISTINCT Country, Region, City FROM dbo.Customers)AS UA
GROUP BY Country, Region, City
HAVING COUNT(*) = 2;

```

The derived table query performs a UNION ALL between distinct rows from both inputs. The outer query groups the unified result by *Country*, *Region* and *City*, and it returns only groups that have two occurrences. In other words, the query returns only distinct rows that appear in both inputs, which is how INTERSECT DISTINCT is defined.

In SQL Server 2005, you simply use the INTERSECT operator as follows:

```

SELECT Country, Region, City FROM dbo.Employees
INTERSECT
SELECT Country, Region, City FROM dbo.Customers;

```

INTERSECT ALL

Like EXCEPT ALL, INTERSECT ALL also considers multiple occurrences of rows. If a row *R* appears *n* times in one input table and *m* times in the other, it should appear *MIN(n, m)* times in the result.

The techniques to achieve INTERSECT ALL have many similarities to the techniques used to achieve EXCEPT ALL. For example, here's a pre—SQL Server 2005 solution to achieve INTERSECT ALL:

```
SELECT Country, Region, City
FROM (SELECT Country, Region, City, MIN(Cnt) AS MinCnt
      FROM (SELECT Country, Region, City, COUNT(*) AS Cnt
            FROM dbo.Employees
            GROUP BY Country, Region, City

            UNIONALL

            SELECT Country, Region, City, COUNT(*)
            FROM dbo.Customers
            GROUP BY Country, Region, City) AS UA
      GROUP BY Country, Region, City
      HAVING COUNT(*) > 1) AS D
JOIN dbo.Nums
  ON n <= MinCnt;
```

UA has the UNION ALL of distinct rows from each input along with the number of occurrences they had in the source. The query against UA that generates the derived table D groups the rows by *Country*, *Region*, and *City*. The HAVING clause filters only rows that appeared in both inputs (*COUNT(*) > 1*), returning their minimum count (*MinCnt*). This is the number of times the row should appear in the output. To generate that many copies, the outer query joins D with Nums based on *n <= MinCnt*.

The solution to INTERSECT ALL in SQL Server 2005 is identical to the one for EXCEPT ALL except for one obvious difference—the use of the INTERSECT operator instead of EXCEPT:

```
WITH INTERSECT_ALL
AS
(
  SELECT
    ROW_NUMBER()
      OVER(PARTITION BY Country, Region, City
           ORDER BY Country, Region, City) AS rn,
    Country, Region, City
  FROM dbo.Employees

  INTERSECT

  SELECT
    ROW_NUMBER()
      OVER(PARTITION BY Country, Region, City
           ORDER BY Country, Region, City) AS rn,
    Country, Region, City
  FROM dbo.Customers
)
SELECT Country, Region, City
FROM INTERSECT_ALL;
```

Precedence of Set Operations

The INTERSECT set operation has a higher precedence than the others. In a query that mixes multiple set operations, INTERSECT is evaluated first. Other than that, set operations are evaluated from left to right. The exception is that parentheses are always first in precedence, so by using parentheses you have full control of the logical order of evaluation of set operations.

For example, in the following query INTERSECT is evaluated first even though it appears second:

```
SELECT Country, Region, City FROM dbo.Suppliers
EXCEPT
SELECT Country, Region, City FROM dbo.Employees
INTERSECT
SELECT Country, Region, City FROM dbo.Customers;
```

The meaning of the query is: return supplier cities that do not appear in the intersection of employee cities and customer cities.

However, if you use parentheses, you can change the evaluation order:

```
(SELECT Country, Region, City FROM dbo.Suppliers
EXCEPT
SELECT Country, Region, City FROM dbo.Employees)
INTERSECT
SELECT Country, Region, City FROM dbo.Customers;
```

This query means: return supplier cities that are not employee cities and are also customer cities.

Using INTO with Set Operations

If you want to write a SELECT INTO statement where you use set operations, specify the INTO clause just before the FROM clause of the first input. For example, here's how you populate a temporary table #T with the result of one of the previous queries:

```
SELECT Country, Region, City INTO #T FROM dbo.Suppliers
EXCEPT
SELECT Country, Region, City FROM dbo.Employees
INTERSECT
SELECT Country, Region, City FROM dbo.Customers;
```

Circumventing Unsupported Logical Phases

As I mentioned earlier, logical processing phases other than sorting (joins, filtering, grouping, TOP, and so on) are not allowed directly on the result of a set operation. This limitation can easily be circumvented by using a derived table or a CTE like so:

```
SELECT DISTINCT TOP ...
FROM(<set operation query>) AS D
JOIN | PIVOT | APPLY...
WHERE ...
GROUP BY ...
HAVING...
ORDER BY ...
```

For example, the following query (which generates the output shown in [Table 5-13](#)) tells you how many cities there are in each country covered by customers or employees:

Table 5-13: Number of Cities per Country Covered by Customers or Employees

Country	NumCities
Argentina	1
Austria	2
Belgium	2
Brazil	4
Canada	3
Denmark	2
Finland	2
France	9
Germany	11
Ireland	1
Italy	3
Mexico	1
Norway	1

Poland	1
Portugal	1
Spain	3
Sweden	2
Switzerland	2
UK	2
USA	14
Venezuela	4

```
SELECT Country, COUNT(*) AS NumCities
FROM (SELECT Country, Region, City FROM dbo.Employees
      UNION
      SELECT Country, Region, City FROM dbo.Customers) AS U
GROUP BY Country;
```

In a similar manner, you can circumvent the limitations on the individual queries used as inputs to the set operation. Each input can be written as a simple SELECT query from a derived table or a CTE, where you use the disallowed elements in the derived table or CTE expression...

For example, the following query returns the two most recent orders for employees 3 and 5, generating the output shown in [Table 5-14](#):

Table 5-14: Two Most Recent Orders for Employees 3 and 5

EmployeeID	OrderID	OrderDate
3	11063	1998-04-30 00:00:00.000
3	11057	1998-04-29 00:00:00.000
5	11043	1998-04-22 00:00:00.000
5	10954	1998-03-17 00:00:00.000

```
SELECT EmployeeID, OrderID, OrderDate
FROM (SELECT TOP 2 EmployeeID, OrderID, OrderDate
      FROM dbo.Orders
      WHERE EmployeeID = 3
      ORDER BY OrderDate DESC, OrderID DESC) AS D1

UNION ALL

SELECT EmployeeID, OrderID, OrderDate
FROM (SELECT TOP 2 EmployeeID, OrderID, OrderDate
      FROM dbo.Orders
      WHERE EmployeeID = 5
      ORDER BY OrderDate DESC, OrderID DESC) AS D2;
```

As for the limitation on sorting the individual inputs, suppose you need to sort each input independently. For example, you want to return orders placed by customer ALFKI and also orders handled by employee 3. As for sorting the rows in the output, you want customer ALFKI's orders to appear first, sorted by *OrderDate* descending, and then orders handled by employee 3, sorted by *OrderID* ascending. To achieve this, you create a column (*SortCol*) with the constant 1 for the first input (customer ALFKI), and 2 for the second (employee 3). Create a derived table (call it U) out of the UNION ALL between the two. In the outer query, first sort by *SortCol*, and then by a CASE expression for each set. The CASE expression will return the relevant value based on the source set; otherwise, it returns a NULL, which won't affect sorting. Here's the solution query generating the output (abbreviated) shown in [Table 5-15](#):

Table 5-15: Sorting Each Input Independently (Abbreviated)

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

1	ALFKI	10835	1998-01-15 00:00:00.000
1	ALFKI	10952	1998-03-16 00:00:00.000
3	ALFKI	11011	1998-04-09 00:00:00.000
3	HUNGO	11063	1998-04-30 00:00:00.000
3	NORTS	11057	1998-04-29 00:00:00.000
3	HANAR	11052	1998-04-27 00:00:00.000
3	GOURL	11049	1998-04-24 00:00:00.000
3	CHOPS	11041	1998-04-22 00:00:00.000
3	QUICK	11021	1998-04-14 00:00:00.000
...			

```

SELECT EmployeeID, CustomerID, OrderID, OrderDate
FROM (SELECT 1 AS SortCol, CustomerID, EmployeeID, OrderID, OrderDate
      FROM dbo.Orders
      WHERE CustomerID = N'ALFKI'

      UNION ALL

      SELECT 2 AS SortCol, CustomerID, EmployeeID, OrderID, OrderDate
      FROM dbo.Orders
      WHERE EmployeeID = 3) AS U
ORDER BY SortCol,
      CASE WHEN SortCol = 1 THEN OrderID END,
      CASE WHEN SortCol = 2 THEN OrderDate END DESC;

```

Conclusion

I covered many aspects of joins and set operations and demonstrated new techniques that you might find handy.

Remember that the old-style proprietary syntax for outer joins is not supported any more and will work only under a backward compatibility mode. At the same time, other types of joins that use the ANSI SQL:1989 syntax are fully supported, as this syntax is still part of the standard—although when using the older syntax for inner joins, there's a risk of getting a Cartesian product when you forget to specify a WHERE clause.

SQL Server 2005 introduces native operators for EXCEPT and INTERSECT. It also provides other tools that allow simple solutions for achieving EXCEPT ALL and INTERSECT ALL.