Chapter 3 Query Execution

—Craig Freedman

In this chapter:

The SQL Server query processor consists of two components: the query optimizer and the query execution engine. The query optimizer is responsible for generating good query plans. The query execution engine takes the query plans generated by the query optimizer and, as its name suggests, runs them. Query execution involves many functions, including using the storage engine to retrieve and update data from tables and indexes and implementing operations such as joins and aggregation.

The focus of this chapter is on understanding query behavior by examining the details of your query execution plans. The chapter explains how the SQL Server query processor works, beginning with the basics of query plans and working toward progressively more complex examples.

Query Processing and Execution Overview

To better understand the factors that affect query performance, to understand how to spot potential performance problems with a query plan, and ultimately to learn how to use query optimizer hints to tune individual query plans, we first need to understand how the SQL Server query processor executes queries. In this section, we introduce iterators, one of the most fundamental query execution concepts, discuss how to read and understand query plans, explore some of the most common query execution operators, and learn how SQL Server combines these operators to execute even the most complex queries.

Iterators

SQL Server breaks queries down into a set of fundamental building blocks that we call operators or iterators. Each iterator implements a single basic operation such as scanning data from a table, updating data in a table, filtering or aggregating data, or joining two data sets. In all, there are a few dozen such primitive iterators. Iterators may have no children or may have one, two, or more children and can be combined into trees which we call query plans. By building appropriate query plans, SQL Server can execute any SQL statement. In practice, there are frequently many valid query plans for a given statement. The query optimizer's job is to find the best (for example, the cheapest or fastest) query plan for a given statement.

An iterator reads input rows either from a data source such as a table or from its children (if it has any) and produces output rows, which it returns to its parent. The output rows that an iterator produces depend on the operation that the iterator performs.

All iterators implement the same set of core methods. For example, the *Open* method tells an iterator to prepare to produce output rows, the *GetRow* method requests that an iterator produce a new output row, and the *Close* method indicates that the iterator's parent is through requesting rows. Because all iterators implement the same methods, iterators are independent of one another. That is, an iterator does not need specialized knowledge of its children (if any) or parent. Consequently, iterators can be easily combined in many different ways and into many different query plans.

When SQL Server executes a query plan, control flows down the query tree. That is, SQL Server calls the methods *Open* and *GetRow* on the iterator at the root of the query tree and these methods propagate down through the tree to the leaf iterators. Data flows or more accurately is pulled up the tree when one iterator calls another iterator's *GetRow* method.

To understand how iterators work, let's look at an example. Most of the examples in this chapter, including the following example, are based on an extended version of the *Northwind* database, called *Northwind2*. You can download a script to build *Northwind2* from the book's companion Web site. Consider this query:

```
SELECT COUNT(*) FROM [Orders]
```

The simplest way to execute this query is to scan each row in the *Orders* table and count the rows. SQL Server uses two iterators to achieve this result: one to scan the rows in the *Orders* table and another to count them, as illustrated in Figure 3-1.

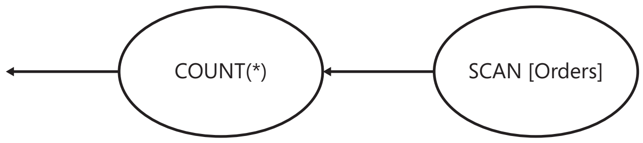


Figure 3-1 Iterators for basic COUNT(*) query

To execute this query plan, SQL Server calls Open on the root iterator in the plan which in this example is the COUNT(*) iterator. The COUNT(*) iterator performs the following tasks in the *Open* method:

1. Call Open on the scan iterator, which readies the scan to produce rows;

2. Call GetRow repeatedly on the scan iterator, counting the rows returned, and stopping only when GetRow indicates that it has returned all of the rows; and
3. Call Close on the scan iterator to indicate that it is done getting rows.

Note COUNT(*) is actually implemented by the stream aggregate iterator, which we will describe in more detail later in this chapter.

Thus, by the time the COUNT(*) iterator returns from Open, it has already calculated the number of rows in the *Orders* table. To complete execution SQL Server calls GetRow on the COUNT(*) iterator and returns this result. [Technically, SQL Server calls GetRow on the COUNT(*) iterator one more time since it does not know that the COUNT(*) iterator produces only a single row until it tries to retrieve a second row. In response to the second GetRow call, the COUNT(*) iterator returns that it has reached the end of the result set.]

Note that the COUNT(*) iterator neither cares nor needs to know that it is counting rows from a scan iterator; it will count rows from any subtree that SQL Server puts below it, regardless of how simple or complex the subtree may be.

Properties of Iterators

Three important properties of iterators can affect query performance and are worth special attention. These properties are memory consumption, nonblocking vs. blocking, and dynamic cursor support.

Memory Consumption

All iterators require some small fixed amount of memory to store state, perform calculations, and so forth. SQL Server does not track this fixed memory or try to reserve this memory before executing a query. When SQL Server caches an executable plan, it caches this fixed memory so that it does not need to allocate it again and to speed up subsequent executions of the cached plan.

However, some iterators, referred to as memory-consuming iterators, require additional memory to execute. This additional memory is used to store row data. The amount of memory required by a memory-consuming operator is generally proportional to the number of rows processed. To ensure that the server does not run out of memory and that queries containing memory-consuming iterators do not fail, SQL Server estimates how much memory these queries need and reserves a memory grant before executing such a query.

Memory-consuming iterators can affect performance in a few ways.

1. Queries with memory-consuming iterators may have to wait to acquire the necessary memory grant and cannot begin execution if the server is executing other such queries

and does not have enough available memory. This waiting can directly affect performance by delaying execution.

2. If too many queries are competing for limited memory resources, the server may suffer from reduced concurrency and/or throughput. This impact is generally not a major issue for data warehouses but is undesirable in OLTP (Online Transaction Processing) systems.
3. If a memory-consuming iterator requests too little memory, it may need to spill data to disk during execution. Spilling can have a significant adverse impact on the query and system performance because of the extra I/O overhead. Moreover, if an iterator spills too much data, it can run out of disk space on *tempdb* and fail.

The primary memory-consuming iterators are sort, hash join, and hash aggregation.

Nonblocking vs. Blocking Iterators

Iterators can be classified into two categories:

1. Iterators that consume input rows and produce output rows at the same time (in the *GetRow* method). We often refer to these iterators as "nonblocking."
2. Iterators that consume all input rows (generally in the *Open* method) before producing any output rows. We refer to these iterators as "blocking" or "stop-and-go."

The compute scalar iterator is a simple example of a nonblocking iterator. It reads an input row, computes a new output value using the input values from the current row, immediately outputs the new value, and continues to the next input row.

The sort iterator is a good example of a blocking iterator. The sort cannot determine the first output row until it has read and sorted all input rows. (The last input row could be the first output row; there is no way to know without first consuming every row.)

Blocking iterators often, but not always, consume memory. For example, as we just noted sort is both memory consuming and blocking. On the other hand, the COUNT(*) example, which we used to introduce the concept of iterators, does not consume memory and yet is blocking. It is not possible to know the number of rows without reading and counting them all.

If an iterator has two children, the iterator may be blocking with respect to one and nonblocking with respect to the other. Hash join (which we'll discuss later in this chapter) is a good example of such an iterator.

Nonblocking iterators are generally optimal for OLTP queries where response time is important. They are often especially desirable for TOP N queries where N is small. Since the goal is to return the first few rows as quickly as possible, it helps to avoid blocking iterators, which might process more data than necessary before returning the first rows. Nonblocking iterators can also be useful when evaluating an EXISTS subquery, where it again helps to avoid processing more data than necessary to conclude that at least one output row exists.

Dynamic Cursor Support

The iterators used in a dynamic cursor query plan have special properties. Among other things, a dynamic cursor plan must be able to return a portion of the result set on each fetch request, must be able to scan forward or backward, and must be able to acquire scroll locks as it returns rows. To support this functionality, an iterator must be able to save and restore its state, must be able to scan forward or backward, must process one input row for each output row it produces, and must be nonblocking. Not all iterators have all of these properties.

For a query to be executed using a dynamic cursor, the optimizer must be able to find a query plan that uses only iterators that support dynamic cursors. It is not always possible to find such a plan. Consequently, some queries cannot be executed using a dynamic cursor. For example, queries that include a GROUP BY clause inherently violate the one input row for each output row requirement. Thus, such queries can never be executed using a dynamic cursor.

Reading Query Plans

To better understand what the query processor is doing, we need a way to look at query plans. SQL Server 2005 has several different ways of displaying a query plan, and we refer to all these techniques collectively as "the showplan options."

Query Plan Options

SQL Server 2005 supports three showplan options: graphical, text, and XML. Graphical and text were available in prior versions of SQL Server; XML is new to SQL Server 2005. Each showplan option outputs the same query plan. The difference between these options is how the information is formatted, the level of detail included, how we read it, and how we can use it.

Graphical Plans

The graphical showplan option uses visually appealing icons that correspond to the iterators in the query plan. The tree structure of the query plan is clear. Arrows represent the data flow between the iterators. ToolTips provide detailed help, including a description of and statistical data on each iterator; this includes estimates of the number of rows generated by each operator (that is, the cardinality estimates), the average row size, and the cost of the operator. In SQL Server 2005, the Management Studio Properties window includes even more detailed information about each operator and about the overall query plan. Much of this data is new and was not available in SQL Server 2000. For example, the Properties window displays the SET options (such as ARITHABORT and ANSI_NULLS) used during the compilation of the plan, parameter and variable values used during optimization and at execution time, thread level execution statistics for parallel plans, the degree of parallelism for parallel plans, the size of the memory grant if any, the size of the cached query plan, requested and actual cursor types, information about query optimization hints, and information on missing indexes.

SQL Server 2005 SP2 adds compilation time (both elapsed and CPU time) and memory. Some of the available data varies from plan type to plan type and from operator to operator.

Generally, graphical plans give a good view of the big picture, which makes them especially useful for beginners and even for experienced users who simply want to browse plans quickly. On the other hand, some very large query plans are so large that they can only be viewed either by scaling the graphics down to a point where the icons are hard to read or by scrolling in two dimensions.

We can generate graphical plans using Management Studio in SQL Server 2005 (or using Query Analyzer in SQL Server 2000). Management Studio also supports saving and reloading graphical plans in files with a .sqlplan extension. In fact, the contents of a .sqlplan file are really just an XML plan and the same information is available in both graphical and XML plans. In prior versions of SQL Server, there is no way to save graphical plans (other than as an image file).

Text Plans

The text showplan option represents each iterator on a separate line. SQL Server uses indentation and vertical bars ("|" characters) to show the child–parent relationship between the iterators in the query tree. There are no explicit arrows, but data always flows up the plan from a child to a parent. Once you understand how to read it, text plans are often easier to read—especially when big plans are involved. Text plans can also be easier than graphical plans to save, manipulate, search, and/or compare, although many of these benefits are greatly diminished if not eliminated with the introduction of XML plans in SQL Server 2005.

There are two types of text plans. You can use SET SHOWPLAN_TEXT ON to display just the query plan. You can use SET SHOWPLAN_ ALL ON to display the query plan along with most of the same estimates and statistics included in the graphical plan ToolTips and Properties windows.

XML Plans

The XML showplan option is new to SQL Server 2005. It brings together many of the best features of text and graphical plans. The ability to nest XML elements makes XML a much more natural choice than text for representing the tree structure of a query plan. XML plans comply with a published XSD schema (*http://schemas.microsoft.com/sqlserver/2004/07/showplan/showplanxml.xsd*) and, unlike text and graphical plans, are easy to search and process programmatically using any standard XML tools. You can even save XML plans in a SQL Server 2005 XML column, index them, and query them using SQL Server 2005's built-in XQuery functionality. Moreover, while compared with text plans the native XML format is more challenging to read directly, as noted previously, Management Studio can save graphical showplan output as XML plan files (with the .sqlplan extension) and can load XML plan files (again with the .sqlplan extension) and display them graphically.

XML plans contain all of the information available in SQL Server 2000 via either graphical or text plans. In addition, XML plans include the same detailed new information mentioned previously that is available using graphical plans and the Management Studio Properties window. XML plans are also the basis for the new USEPLAN query hint described in Chapters 4 and 5.

The XML plan follows a hierarchy of a batch element, a statement element, and a query plan element (*<QueryPlan>*). If a batch or procedure contains multiple statements, the XML plan output for that batch or procedure will contain multiple query plans. Within the query plan element is a series of relational operator elements (*<RelOp>*). There is one relational operator element for each iterator in the query plan, and these elements are nested according to the tree structure of the query plan. Like the other showplan options, each relational operator element includes cost estimates and statistics, as well as some operator-specific information.

Estimated vs. Actual Query Plans

We can ask SQL Server to output a plan (for any showplan option–graphical, text, or XML) with or without actually running a query.

We refer to a query plan generated without executing a query as the "estimated execution plan" as SQL Server may choose to recompile the query (recompiles may occur for a variety of reasons) and may generate a different query plan at execution time. The estimated execution plan is useful for a variety of purposes, such as viewing the query plan of a long-running query without waiting for it to complete; viewing the query plan for an insert, update, or delete statement without altering the state of the database or acquiring any locks; or exploring the effect of various optimization hints on a query plan without actually running the query. The estimated execution plan includes cardinality, row size, and cost estimates.

Tip The estimated costs reported by the optimizer are intended as a guide to compare the anticipated relative cost of various operators within a single query plan or the relative cost of two different plans. These estimates are unitless and are not meant to be interpreted in any absolute sense such as milliseconds or seconds.

We refer to a query plan generated after executing a query as the "actual execution plan." The actual execution plan includes the same information as the estimated execution plan plus the actual row counts and the actual number of executions for each operator. By comparing the estimated and actual row counts, we can identify cardinality estimation errors, which may lead to other plan issues. XML plans include even more information, such as actual parameter and variable values at execution time; the memory grant and degree of parallelism if appropriate; and thread level row, execution, rewind, and rebind counts. (We cover rewinds and rebinds later in this chapter.)

Tip The actual execution plan includes the same cost estimates as the estimated execution plan. Although SQL Server actually executes the query plan while generating the actual execution plan, these cost estimates are still the same estimates generated by the optimizer and do not reflect the actual execution cost.

There are several Transact-SQL commands that we can use to collect showplan option output when running ad hoc queries from SQL Server Management Studio or from the SQLCMD command line utility. These commands allow us to collect both text and XML plans, as well as estimated and actual plans. Table 3-1 lists all of the available SET commands to enable showplan options.

Table 3-1 SET Commands for Displaying Query Plans

	Command	**Execute Query?**	**Include Estimated Row Counts & Stats**	**Include Actual Row Counts & Stats**
Text Plan	SET SHOWPLAN_TEXT ON	No	No	No
	SET SHOWPLAN_ALL ON	No	Yes	No
	SET STATISTICS PROFILE ON	Yes	Yes	Yes
XML Plan	SET SHOWPLAN_XML ON	No	Yes	No
	SET STATISTICS PROFILE XML	Yes	Yes	Yes

We can also collect all forms of query plans using SQL Trace and XML plans using Dynamic Management Views (DMVs) (which are new to SQL Server 2005). These options are especially useful when analyzing applications in which you do not have access to the source code. Obtaining plan information from traces is discussed in Chapter 2, "Tracing and Profiling." The DMVs that contain plan information are discussed in Chapter 5, "Plan Caching and Recompilation."

Query Plan Display Options

Let's compare the various ways of viewing query plans. As an example, consider the following query:

```
DECLARE @Country nvarchar(15)
SET @Country = N'USA'
SELECT O.[CustomerId], MAX(O.[Freight]) AS MaxFreight
FROM [Customers] C JOIN [Orders] O
   ON C.[CustomerId] = O.[CustomerId]
WHERE C.[Country] = @Country
GROUP BY O.[CustomerId]
OPTION (OPTIMIZE FOR (@Country = N'UK'))
```

The graphical plan for this query is shown in Figure 3-2.

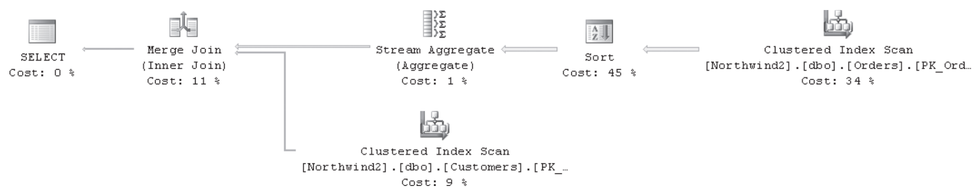
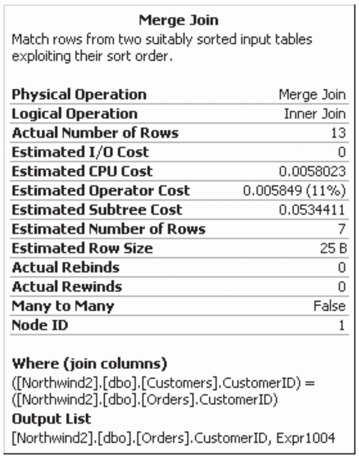


Figure 3-2 A graphical execution plan

Do not be too concerned at this point with understanding how the operators in this query plan actually function. Later in this chapter, we will delve into the details of the various operators. For now, simply observe how SQL Server combines the individual operators together in a tree structure. Notice that the clustered index scans are leaf operators and have no children, the sort and stream aggregate operators have one child each, and the merge join operator has two children. Also, notice how the data flows as shown by the arrows from the leaf operators on the right side of the plan to the root of the tree on the left side of the plan.

Figure 3-3 shows the ToolTip information and Figure 3-4 shows the Properties window from the actual (runtime) plan for the *merge join* operator. The ToolTip and Properties window show additional information about the operator, the optimizer's cost and cardinality estimates, and the actual number of output rows.



Merge Join

Match rows from two suitably sorted input tables exploiting their sort order.

Physical Operation	Merge Join
Logical Operation	Inner Join
Actual Number of Rows	13
Estimated I/O Cost	0
Estimated CPU Cost	0.0058023
Estimated Operator Cost	0.005849 (11%)
Estimated Subtree Cost	0.0534411
Estimated Number of Rows	7
Estimated Row Size	25 B
Actual Rebinds	0
Actual Rewinds	0
Many to Many	False
Node ID	1

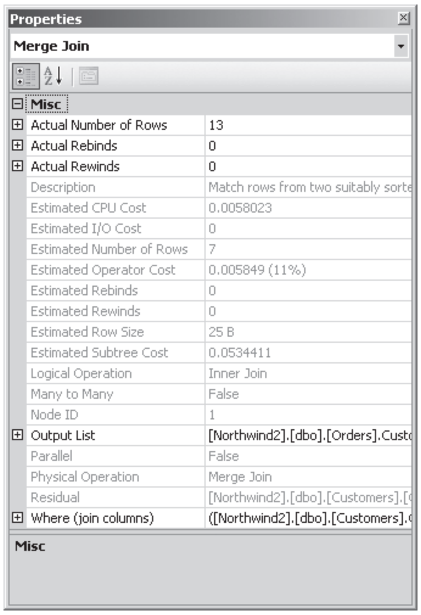
Where (join columns)

([Northwind2].[dbo].[Customers].CustomerID) = ([Northwind2].[dbo].[Orders].CustomerID)

Output List

[Northwind2].[dbo].[Orders].CustomerID, Expr1004

Figure 3-3 ToolTip for merge join operator in a graphical plan



Properties

Merge Join

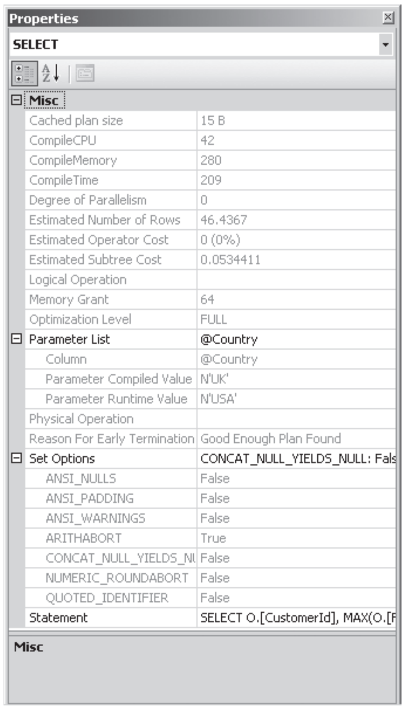
Misc

Actual Number of Rows	13
Actual Rebinds	0
Actual Rewinds	0
Description	Match rows from two suitably sorte
Estimated CPU Cost	0.0058023
Estimated I/O Cost	0
Estimated Number of Rows	7
Estimated Operator Cost	0.005849 (11%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	25 B
Estimated Subtree Cost	0.0534411
Logical Operation	Inner Join
Many to Many	False
Node ID	1
Output List	[Northwind2].[dbo].[Orders].Custo
Parallel	False
Physical Operation	Merge Join
Residual	[Northwind2].[dbo].[Customers].[
Where (join columns)	([Northwind2].[dbo].[Customers].

Misc

Figure 3-4 Properties window for merge join operator

Figure 3-5 shows the Properties window for the SELECT icon at the root of the plan. Note that it includes query-wide information such as the SET options used during compilation, the compilation time and memory, the cached plan size, the degree of parallelism, the memory grant, and the parameter and variable values used during compilation and execution. We will discuss the meaning of these fields as part of the XML plan example below. Keep in mind that a variable and a parameter are very different elements and the difference will be discussed in detail in Chapter 5. However, the various query plans that we will examine use the term *parameter* to refer to either variables or parameters.


Properties

SELECT

Misc	
Cached plan size	15 B
CompileCPU	42
CompileMemory	280
CompileTime	209
Degree of Parallelism	0
Estimated Number of Rows	46.4367
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0534411
Logical Operation	
Memory Grant	64
Optimization Level	FULL
Parameter List	@Country
Column	@Country
Parameter Compiled Value	N'UK'
Parameter Runtime Value	N'USA'
Physical Operation	
Reason For Early Termination	Good Enough Plan Found
Set Options	CONCAT_NULL_YIELDS_NULL: Fals
ANSI_NULLS	False
ANSI_PADDING	False
ANSI_WARNINGS	False
ARITHABORT	True
CONCAT_NULL_YIELDS_NU	False
NUMERIC_ROUNDABORT	False
QUOTED_IDENTIFIER	False
Statement	SELECT O.[CustomerId], MAX(O.[F

Misc

Figure 3-5 Properties window for SELECT at the top of a query plan

Now let's consider the same query plan by looking at the output of SET SHOWPLAN_TEXT ON. Here is the text plan showing the query plan only:

```
|--Merge Join(Inner Join, MERGE:([O].[CustomerID])=([C].[CustomerID]), RESIDUAL:(...))
     |--Stream Aggregate(GROUP BY:([O].[CustomerID])
        DEFINE:([Expr1004]=MAX([O].[Freight])))
     |    |--Sort(ORDER BY:([O].[CustomerID] ASC))
     |        |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O]))
     |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]),
        WHERE:([C].[Country]=[@Country]) ORDERED FORWARD)
```

Note This plan and all of the other text plan examples in this chapter and in Chapter 4 "Troubleshooting Query Performance," have been edited for brevity and to improve clarity. For instance, the database and schema name of objects have been removed from all plans. In some cases, lines have been wrapped, where they wouldn't normally wrap in the output.

Notice how, while there are no icons or arrows, this view of the plan has precisely the same operators and tree structure as the graphical plan. Recall that each line represents one operator—the equivalent of one icon in the graphical plan—and the vertical bars (the "|" characters) link each operator to its parent and children.

The output of SET SHOWPLAN_ALL ON includes the same plan text but, as noted previously, also includes additional information including cardinality and cost estimates. The SET STATISTICS PROFILE ON output includes actual row and operator execution counts, in addition to all of the other information.

Finally, here is a highly abbreviated version of the SET STATISTICS XML ON output for the same query plan. Notice how we have the same set of operators in the XML version of the plan as we did in the graphical and text versions. Also observe how the child operators are nested within the parent operator's XML element. For example, the merge join has two children and, thus, there are two relational operator elements nested within the merge join's relational operator element.

```
<StmtSimple StatementText=
     "SELECT O.[CustomerId], MAX(O.[Freight]) as MaxFreight
     FROM [Customers] C JOIN [Orders] O
            ON C.[CustomerId] = O.[CustomerId
     WHERE C.[Country] = @Country
     GROUP BY O.[CustomerId]
     OPTION (OPTIMIZE FOR (@Country = N'UK'))"...>
  <StatementSetOptions QUOTED_IDENTIFIER="false" ARITHABORT="true"
                       CONCAT_NULL_YIELDS_NULL="false" ANSI_NULLS="false"
                       ANSI_PADDING="false" ANSI_WARNINGS="false"
                       NUMERIC_ROUNDABORT="false" />
  <QueryPlan DegreeOfParallelism="0" MemoryGrant="64" CachedPlanSize="15"
             CompileTime="20" CompileCPU="20" CompileMemory="280">
    <RelOp NodeId="1" PhysicalOp="Merge Join" LogicalOp="Inner Join"...>
      <Merge ManyToMany="0">
        <RelOp NodeId="2" PhysicalOp="Stream Aggregate" LogicalOp="Aggregate"...>
          <StreamAggregate>
            <RelOp NodeId="3" PhysicalOp="Sort" LogicalOp="Sort"...>
              <MemoryFractions Input="1" Output="1" />
              <Sort Distinct="0">
                <RelOp NodeId="4" PhysicalOp="Clustered Index Scan"
                                  LogicalOp="Clustered Index Scan"...>
                  <IndexScan Ordered="0" ForcedIndex="0" NoExpandHint="0">
                    <Object Database="[Northwind2]" Schema="[dbo]" Table="[Orders]"
                                  Index="[PK_Orders]" Alias="[O]" />
                  </IndexScan>
                </RelOp>
              </Sort>
            </RelOp>
          </StreamAggregate>
        </RelOp>
        <RelOp NodeId="8" PhysicalOp="Clustered Index Scan"
                          LogicalOp="Clustered Index Scan"...>
            <IndexScan Ordered="1" ScanDirection="FORWARD" ForcedIndex="0"
                      NoExpandHint="0">
              <Object Database="[Northwind2]" Schema="[dbo]" Table="[Customers]"
                      Index="[PK_Customers]" Alias="[C]" />
                <Predicate>
                  <ScalarOperator ScalarString="[Northwind2].[dbo].[Customers].[Country]
                                        as [C].[Country]=[@Country]">
                  </ScalarOperator>
                </Predicate>
            </IndexScan>
          </RelOp>
        </Merge>
      </RelOp>
      <ParameterList>
        <ColumnReference Column="@Country" ParameterCompiledValue="N'UK'"
                 ParameterRuntimeValue="N'USA'" />
      </ParameterList>
    </QueryPlan>
</StmtSimple>
```

There are some other elements worth pointing out:

- The <*StmtSimple*> element includes a *StatementText* attribute, which as one might expect, includes the original statement text. Depending on the statement type, the <*StmtSimple*> element may be replaced by another element such as <*StmtCursor*>.
- The <*StatementSetOptions*> element includes attributes for the various SET options.
- The <*QueryPlan*> element includes the following attributes:
    - *DegreeOfParallelism:* The number of threads per operator for a parallel plan. A value of zero or one indicates a serial plan. This example is a serial plan.
    - *MemoryGrant*: The total memory granted to run this query in 2-Kbyte units. (The memory grant unit is documented as Kbytes in the showplan schema but is actually reported in 2-Kbyte units.) This query was granted 128 Kbytes.
    - *CachedPlanSize:* The amount of plan cache memory (in Kbytes) consumed by this query plan.
    - *CompileTime* and *CompileCPU*: The elapsed and CPU time (in milliseconds) used to compile this plan. (These attributes are new in SQL Server 2005 SP2.)
    - *CompileMemory*: The amount of memory (in Kbytes) used while compiling this query. (This attribute is new in SQL Server 2005 SP2.)
- The <*QueryPlan*> element also includes a <*ParameterList*> element, which includes the compile time and run-time values for each parameter and variable. In this example, there is just the one @Country variable.
- The <*RelOp*> element for each memory-consuming operator (in this example just the sort) includes a <*MemoryFractions*> element, which indicates the portion of the total memory grant used by that operator. There are two fractions. The input fraction refers to

the portion of the memory grant used while the operator is reading input rows. The output fraction refers to the portion of the memory grant used while the operator is producing output rows. Generally, during the input phase of an operator's execution, it must share memory with its children; during the output phase of an operator's execution, it must share memory with its parent. Since in this example, the sort is the only memory-consuming operator in the plan, it uses the entire memory grant. Thus, the fractions are both one.

Although they have been truncated from the above output, each of the relational operator elements includes additional attributes and elements with all of the estimated and run-time statistics available in the graphical and text query plan examples:

```
<RelOp NodeId="1" PhysicalOp="Merge Join" LogicalOp="Inner Join"
                  EstimateRows="7" EstimateIO="0"
                  EstimateCPU="0.0058023" AvgRowSize="25"
                  EstimatedTotalSubtreeCost="0.0534411" Parallel="0"
                  EstimateRebinds="0" EstimateRewinds="0">
   <RunTimeInformation>
     <RunTimeCountersPerThread Thread="0" ActualRows="13"
                    ActualEndOfScans="1" ActualExecutions="1" />
   </RunTimeInformation>
   ...
</RelOp>
```

Note Most of the examples in this chapter display the query plan in text format, obtained with SET SHOWPLAN_TEXT ON. Text format is more compact and easier to read than XML format and also includes more detail than screenshots of plans in graphical format. However, in some cases it is important to observe the "shape" of a query plan, and we will be showing you some examples of graphical plans. If you prefer to see plans in a format other than the one supplied in this chapter, you can download the code for the queries in this chapter from the companion Web site, and display the plans in the format of your choosing using your own SQL Server Management Studio.

Analyzing Plans

To really understand query plans and to really be able to spot, fix, or work around problems with query plans, we need a solid understanding of the query operators that make up these plans. All in all, there are too many operators to discuss them in one chapter. Moreover, there are innumerable ways to combine these operators into query plans. Thus, in this section, we focus on understanding the most common query operators—the most basic building blocks of query execution—and give some insight into when and how SQL Server uses them to construct a variety of interesting query plans. Specifically, we will look at scans and seeks, joins, aggregations, unions, a selection of subquery plans, and parallelism. With an understanding of how these basic operators and plans work, it is possible to break down and understand much bigger and more complex query plans.

Scans and Seeks

Scans and seeks are the iterators that SQL Server uses to read data from tables and indexes. These iterators are among the most fundamental ones that SQL Server supports. They appear in nearly every query plan. It is important to understand the difference between scans and seeks: a scan processes an entire table or the entire leaf level of an index, whereas a seek efficiently returns rows from one or more ranges of an index based on a predicate.

Let's begin by looking at an example of a scan. Consider the following query:

```
SELECT [OrderId] FROM [Orders] WHERE [RequiredDate] = '1998-03-26'
```

We have no index on the *RequiredDate* column. As a result, SQL Server must read every row of the *Orders* table, evaluate the predicate on *RequiredDate* for each row, and, if the predicate is true (that is, if the row qualifies), return the row.

To maximize performance, whenever possible, SQL Server evaluates the predicate in the scan iterator. However, if the predicate is too complex or too expensive, SQL Server may evaluate it in a separate filter iterator. The predicate appears in the text plan with the WHERE keyword or in the XML plan with the <Predicate> tag. Here is the text plan for the above query:

```
|--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]),
   WHERE:([Orders].[RequiredDate]='1998-03-26'))
```

Figure 3-6 illustrates a scan:

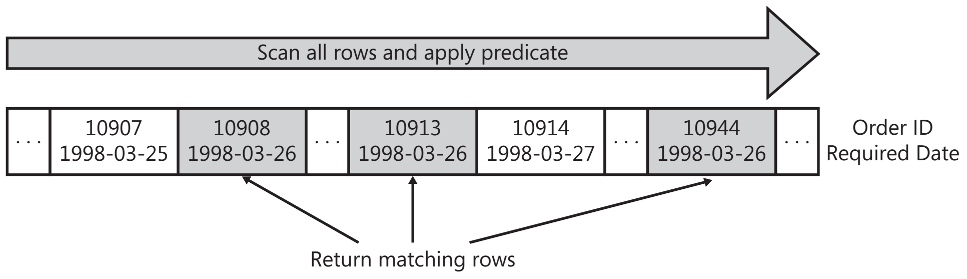


Figure 3-6 A scan operation examines all the rows in all the pages of a table

Since a scan touches every row in the table whether or not it qualifies, the cost is proportional to the total number of rows in the table. Thus, a scan is an efficient strategy if the table is small or if many of the rows qualify for the predicate. However, if the table is large and if most of the rows do *not* qualify, a scan touches many more pages and rows and performs many more I/Os than is necessary.

Now let's look at an example of an index seek. Suppose we have a similar query, but this time the predicate is on the *OrderDate* column on which we do have an index:

```
SELECT [OrderId] FROM [Orders] WHERE [OrderDate] = '1998-02-26'
```

This time SQL Server is able to use the index to navigate directly to those rows that satisfy the predicate. In this case, we refer to the predicate as a seek predicate. In most cases, SQL Server does not need to evaluate the seek predicate explicitly; the index ensures that the seek operation only returns rows that qualify. The seek predicate appears in the text plan with the SEEK keyword or in the XML plan with the <SeekPredicates> tag. Here is the text plan for this example:

```
|--Index Seek(OBJECT:([Orders].[OrderDate]),
   SEEK:([Orders].[OrderDate]=CONVERT_IMPLICIT(datetime,[@1],0)) ORDERED FORWARD)
```

Note Notice that SQL Server autoparameterized the query by substituting the parameter @1 for the literal date.

Figure 3-7 illustrates an index seek:

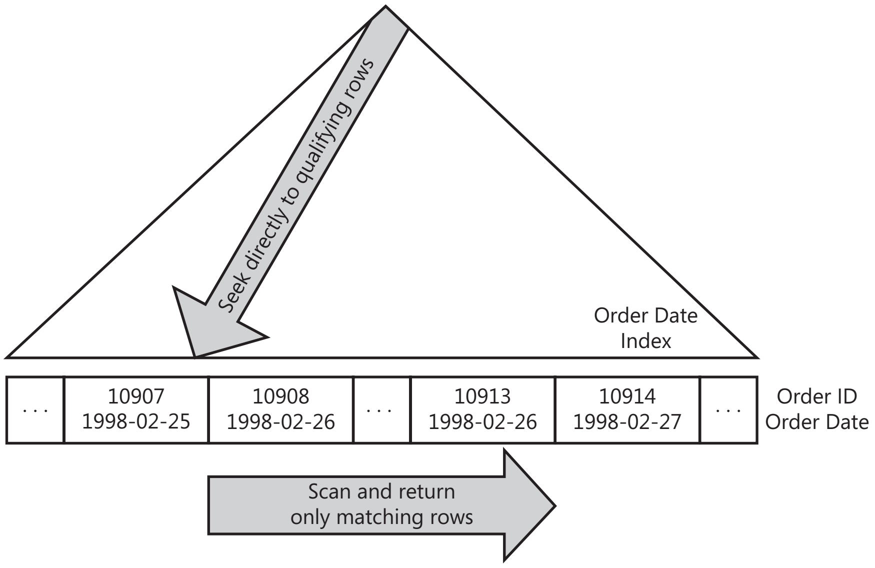


Figure 3-7 An index seek starts at the root and navigates to the leaf to find qualifying rows

Since a seek only touches rows that qualify and pages that contain these qualifying rows, the cost is proportional to the number of qualifying rows and pages rather than to the total number of rows in the table. Thus, a seek is generally a more efficient strategy if we have a highly selective seek predicate; that is, if we have a seek predicate that eliminates a large fraction of the table.

SQL Server distinguishes between scans and seeks as well as between scans on heaps (an object with no clustered index), scans on clustered indexes, and scans on nonclustered indexes. Table 3-2 shows how all of the valid combinations appear in plan output.

Table 3-2 Scan and Seek Operators as They Appear in a Query Plan

	Scan	Seek
Heap	Table Scan	
Clustered Index	Clustered Index Scan	Clustered Index Seek
Nonclustered Index	Index Scan	Index Seek

Seekable Predicates and Covered Columns

Before SQL Server can perform an index seek, it must determine whether the keys of the index are suitable for evaluating a predicate in the query. We refer to a predicate that may be used as the basis for an index seek as a "seekable predicate." SQL Server must also determine whether the index contains or "covers" the set of the columns that are referenced by the query. The following discussion explains how to determine which predicates are seekable, which predicates are not seekable, and which columns an index covers.

Single-Column Indexes

Determining whether a predicate can be used to seek on a single-column index is fairly straightforward. SQL Server can use single-column indexes to answer most simple comparisons including equality and inequality (greater than, less than, etc.) comparisons. More complex expressions, such as functions over a column and LIKE predicates with a leading wildcard character, will generally prevent SQL Server from using an index seek.

For example, suppose we have a single-column index on a column *Col1*. We can use this index to seek on these predicates:

- [Col1] = 3.14
- [Col1] > 100
- [Col1] BETWEEN 0 AND 99
- [Col1] LIKE 'abc%'
- [Col1] IN (2, 3, 5, 7)

However, we cannot use the index to seek on these predicates:

- ABS([Col1]) = 1
- [Col1] + 1 = 9
- [Col1] LIKE '%abc'

Composite Indexes

Composite, or multicolumn, indexes are slightly more complex. With a composite index, the order of the keys matters. It determines the sort order of the index, and it affects the set of seek predicates that SQL Server can evaluate using the index.

For an easy way to visualize why order matters, think about a phone book. A phone book is like an index with the keys (last name, first name). The contents of the phone book are sorted by last name, and we can easily look someone up if we know their last name. However, if we have only a first name, it is very difficult to get a list of people with that name. We would need another phone book sorted on first name.

In the same way, if we have an index on two columns, we can only use the index to satisfy a predicate on the second column if we have an equality predicate on the first column. Even if we cannot use the index to satisfy the predicate on the second column, we may be able to use it on the first column. In this case, we introduce a "residual" predicate for the predicate on the second column. This predicate is evaluated just like any other scan predicate.

For example, suppose we have a two-column index on columns *Col1* and *Col2*. We can use this index to seek on any of the predicates that worked on the single-column index. We can also use it to seek on these additional predicates:

- [Col1] = 3.14 AND [Col2] = 'pi'
- [Col1] = 'xyzzy' AND [Col2] <= 0

For the next set of examples, we can use the index to satisfy the predicate on column *Col1*, but not on column *Col2*. In these cases, we need a residual predicate for column *Col2*.

- [Col1] > 100 AND [Col2] > 100
- [Col1] LIKE 'abc%' AND [Col2] = 2

Finally, we cannot use the index to seek on the next set of predicates as we cannot seek even on column *Col1*. In these cases, we must use a different index (that is, one where column *Col2* is the leading column) or we must use a scan with a predicate.

- [Col2] = 0
- [Col1] + 1 = 9 AND [Col2] BETWEEN 1 AND 9
- [Col1] LIKE '%abc' AND [Col2] IN (1, 3, 5)

Identifying an Index's Keys

In most cases, the index keys are the set of columns that you specify in the CREATE INDEX statement. However, when you create a nonunique nonclustered index on a table with a clustered index, we append the clustered index keys to the nonclustered index keys if they are not explicitly part of the nonclustered index keys. You can seek on these implicit keys as if you specified them explicitly.

Covered Columns The heap or clustered index for a table (often called the "base table") contains (or "covers") all columns in the table. Nonclustered indexes, on the other hand, contain (or cover) only a subset of the columns in the table. By limiting the set of columns stored in a nonclustered index, SQL Server can store more rows on each page, which saves disk space and improves the efficiency of seeks and scans by reducing the number of I/Os and the number of pages touched. However, a scan or seek of an index can only return the columns that the index covers.

Each nonclustered index covers the key columns that were specified when it was created. Also, if the base table is a clustered index, each nonclustered index on this table covers the clustered index keys regardless of whether they are part of the nonclustered index's key columns. In SQL Server 2005, we can also add additional nonkey columns to a nonclustered index using the INCLUDE clause of the CREATE INDEX statement. Note that unlike index keys, order is not relevant for included columns.

Example of Index Keys and Covered Columns For example, given this schema:

```
CREATE TABLE T_heap (a int, b int, c int, d int, e int, f int)
CREATE INDEX T_heap_a ON T_heap (a)
CREATE INDEX T_heap_bc ON T_heap (b, c)
CREATE INDEX T_heap_d ON T_heap (d) INCLUDE (e)
CREATE UNIQUE INDEX T_heap_f ON T_heap (f)

CREATE TABLE T_clu (a int, b int, c int, d int, e int, f int)
CREATE UNIQUE CLUSTERED INDEX T_clu_a ON T_clu (a)
CREATE INDEX T_clu_b ON T_clu (b)
CREATE INDEX T_clu_ac ON T_clu (a, c)
CREATE INDEX T_clu_d ON T_clu (d) INCLUDE (e)
CREATE UNIQUE INDEX T_clu_f ON T_clu (f)
```

The key columns and covered columns for each index are shown in Table 3-3.

Table 3-3 Key Columns and Covered Columns in a Set of Nonclustered Indexes

Index	Key Columns	Covered Columns
T_heap_a	a	a
T_heap_bc	b, c	b, c
T_heap_d	d	d, e
T_heap_f	f	f
T_clu_a	a	a, b, c, d, e, f
T_clu_b	b, a	a, b
T_clu_ac	a, c	a, c
T_clu_d	d, a	a, d, e
T_clu_f	f	a, f

Note that the key columns for each of the nonclustered indexes on *T_clu* include the clustered index key column *a* with the exception of *T_clu_f*, which is a unique index. *T_clu_ac* includes column *a* explicitly as the first key column of the index, and so the column appears in the index only once and is used as the first key column. The other indexes do not explicitly include column *a*, so the column is merely appended to the end of the list of keys.

Bookmark Lookup

We've just seen how SQL Server can use an index seek to efficiently retrieve data that matches a predicate on the index keys. However, we also know that nonclustered indexes do not cover all of the columns in a table. Suppose we have a query with a predicate on a nonclustered index key that selects columns that are not covered by the index. If SQL Server performs a seek on the nonclustered index, it will be missing some of the required columns. Alternatively, if it performs a scan of the clustered index (or heap), it will get all of the columns, but will touch every row of the table and the operation will be less efficient. For example, consider the following query:

```
SELECT [OrderId], [CustomerId] FROM [Orders] WHERE [OrderDate] = '1998-02-26'
```

This query is identical to the query we used earlier to illustrate an index seek, but this time the query selects two columns: *OrderId* and *CustomerId*. The nonclustered index *OrderDate* only covers the *OrderId* column (which also happens to be the clustering key for the *Orders* table in the *Northwind2* database).

SQL Server has a solution to this problem. For each row that it fetches from the nonclustered index, it can look up the value of the remaining columns (for instance, the *CustomerId* column in our example) in the clustered index. We call this operation a "bookmark lookup." A bookmark is a pointer to the row in the heap or clustered index. SQL Server stores the bookmark for each row in the nonclustered index precisely so that it can always navigate from the nonclustered index to the corresponding row in the base table.

Figure 3-8 illustrates a bookmark lookup from a nonclustered index to a clustered index.

SQL Server 2000 implemented bookmark lookup using a dedicated iterator. The text plan shows us the index seek and bookmark lookup operators, as well as indicating the column used for the seek:

```
|--Bookmark Lookup(BOOKMARK:([Bmk1000]), OBJECT:([Orders]))
     |--Index Seek(OBJECT:([Orders].[OrderDate]),
        SEEK:([Orders].[OrderDate]=Convert([@1])) ORDERED FORWARD)
```

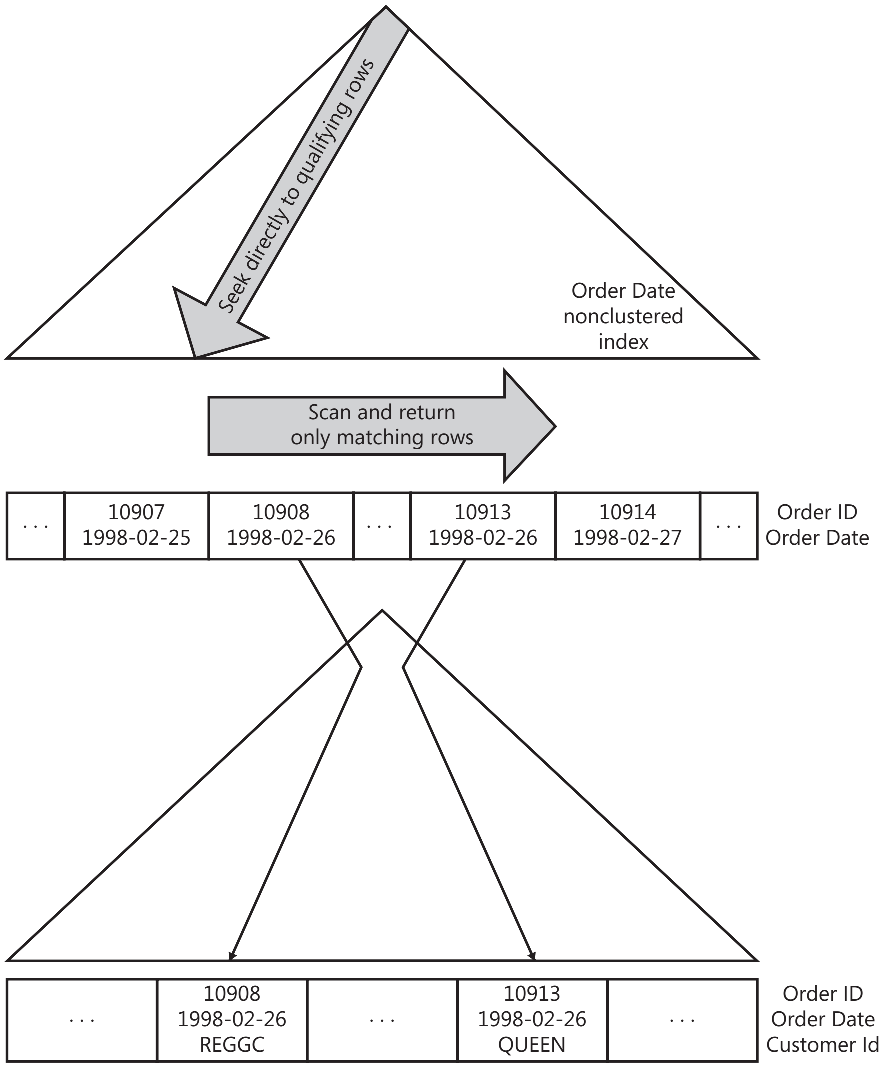


Figure 3-8 A bookmark lookup uses the information from the nonclustered index leaf level to find the row in the clustered index

The graphical plan is shown in Figure 3-9.

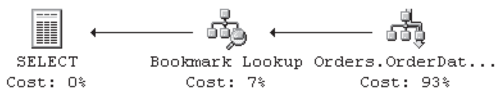


Figure 3-9 Graphical plan for index seek and bookmark lookup in SQL Server 2000

The SQL Server 2005 plan for the same query uses a nested loops join (we will explain the behavior of this operator later in this chapter) combined with a clustered index seek, if the base table is a clustered index, or a RID (row id) lookup if the base table is a heap.

The SQL Server 2005 plans may look different from the SQL Server 2000 plans, but logically they are identical. You can tell that a clustered index seek is a bookmark lookup by the LOOKUP keyword in text plans or by the attribute *Lookup="1"* in XML plans. For example, here is the text plan for the previous query executed on SQL Server 2005:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Orders].[OrderID]))
       |--Index Seek(OBJECT:([Orders].[OrderDate]),
          SEEK:([Orders].[OrderDate]='1998-02-26') ORDERED FORWARD)
       |--Clustered Index Seek(OBJECT:([Orders].[PK_Orders]),
          SEEK:([Orders].[OrderID]=[Orders].[OrderID]) LOOKUP ORDERED FORWARD)
```

In SQL Server 2005 and SQL Server 2005 SP1, a bookmark lookup in graphical plans uses the same icon as any other clustered index seek. We can only distinguish a normal clustered index seek from a bookmark lookup by checking for the Lookup property. In SQL Server 2005 SP2, a bookmark lookup in a graphical plan uses a new Key Lookup icon. This new icon makes the distinction between a normal clustered index seek and a bookmark lookup very clear. However, note that internally there was no change to the operator between SP1 and SP2. Figure 3-10 illustrates the graphical plan in SQL Server 2005. If you're used to looking at SQL Server 2000 plans, you might find it hard to get used to the representation in SQL Server 2005, but as mentioned previously, logically SQL Server is still doing the same work.You might eventually find SQL Server 2005's representation more enlightening, as it makes it clearer that SQL Server is performing multiple lookups into the underlying table.

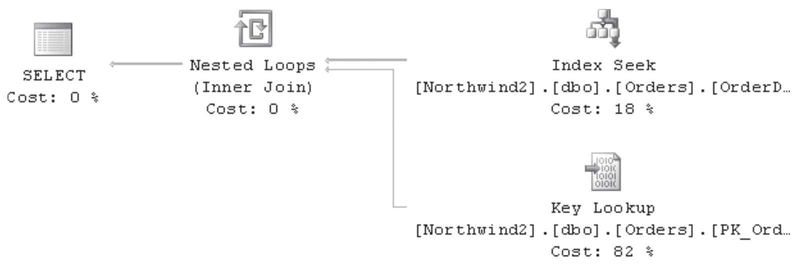


Figure 3-10 Graphical plan for index seek and bookmark lookup in SQL Server 2005 SP2

Bookmark lookup can be used with heaps as well as with clustered indexes, as shown above. In SQL Server 2000, a bookmark lookup on a heap looks the same as a bookmark lookup on a clustered index. In SQL Server 2005, a bookmark lookup on a heap still uses a nested loops join, but instead of a clustered index seek, SQL Server uses a RID lookup operator. A RID lookup operator includes a seek predicate on the heap bookmark, but a heap is not an index and a RID lookup is a not an index seek.

Bookmark lookup is not a cheap operation. Assuming (as is commonly the case) that no correlation exists between the nonclustered and clustered index keys, each bookmark lookup performs a random I/O into the clustered index or heap. Random I/Os are very expensive. When comparing various plan alternatives including scans, seeks, and seeks with bookmark lookups, the optimizer must decide whether it is cheaper to perform more sequential I/Os and touch more rows using an index scan (or an index seek with a less selective predicate) that covers all required columns, or to perform fewer random I/Os and touch fewer rows using a seek with a more selective predicate and a bookmark lookup. Because random I/Os are so much more expensive than sequential I/Os, the cutoff point beyond which a clustered index scan becomes cheaper than an index seek with a bookmark lookup generally involves a surprisingly small percentage of the total table—often just a few percent of the total rows.

Tip In some cases, you can introduce a better plan option by creating a new index or by adding one or more columns to an existing index so as to eliminate a bookmark lookup or change a scan into a seek. In SQL Server 2000, the only way to add columns to an index is to add additional key columns. As noted previously, in SQL Server 2005, you can also add columns using the INCLUDE clause of the CREATE INDEX statement. Included columns are more efficient than key columns. Compared to adding an extra key column, adding an included column uses less disk space and makes searching and updating the index more efficient. Of course, whenever you create new indexes or add new keys or included columns to an existing index, you do consume additional disk space and you do make it more expensive to search and update the index. Thus, you must balance the frequency and importance of the queries that benefit from the new index against the queries or updates that are slower.

Joins

SQL Server supports three physical join operators: nested loops join, merge join, and hash join. We've already seen a nested loops join in the bookmark lookup example. In the following sections, we take a detailed look at how each of these join operators works, explain what logical join types each operator supports, and discuss the performance trade-offs of each join type.

Before we get started, let's put one common myth to rest. There is no "best" join operator, and no join operator is inherently good or bad. We cannot draw any conclusions about a query plan merely from the presence of a particular join operator. Each join operator performs well in the right circumstances and poorly in the wrong circumstances. As we describe each join operator, we will discuss its strengths and weaknesses and the conditions and circumstances under which it performs well.

Nested Loops Join

The nested loops join is the simplest and most basic join algorithm. It compares each row from one table (known as the "outer table") to each row from the other table (known as the "inner table"), looking for rows that satisfy the join predicate.

Note The terms *inner* and *outer* are overloaded; we must infer their meaning from context. "Inner table" and "outer table" refer to the inputs to the join. "Inner join" and "outer join" refer to the semantics of the logical join operations.

We can express the nested loops join algorithm in pseudo-code as:

```
for each row R1 in the outer table
   for each row R2 in the inner table
      if R1 joins with R2
         return (R1, R2)
```

It's the nesting of the loops in this algorithm that gives *nested loops* join its name.

The total number of rows compared and, thus, the cost of this algorithm is proportional to the size of the outer table multiplied by the size of the inner table. Since this cost grows quickly as the size of the input tables grow, in practice the optimizer tries to minimize the cost by reducing the number of inner rows that must be processed for each outer row.

For example, consider this query:

```
SELECT O.[OrderId]
FROM [Customers] C JOIN [Orders] O ON C.[CustomerId] = O.[CustomerId]
WHERE C.[City] = N'London'
```

When we execute this query, we get the following query plan:

```
Rows Executes
46  1       |--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
6   1             |--Index Seek(OBJECT:([Customers].[City] AS [C]),
                        SEEK:([C].[City]=N'London') ORDERED FORWARD)
46  6             |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
                        SEEK:([O].[CustomerID]=[C].[CustomerID]) ORDERED FORWARD)
```

Unlike most of the examples in this chapter, this plan was generated using SET STATISTICS PROFILE ON so that we can see the number of rows and executions for each operator. The outer table in this plan is *Customers* while the inner table is *Orders*. Thus, according to the nested loops join algorithm, SQL Server begins by seeking on the *Customers* table. The join takes one customer at a time and, for each customer, it performs an index seek on the *Orders* table. Since there are six customers, it executes the index seek on the *Orders* table six times. Notice that the index seek on the *Orders* table depends on the *CustomerId,* which comes from the *Customers* table. Each of the six times that SQL Server repeats the index seek on the *Orders* table, *CustomerId* has a different value. Thus, each of the six executions of the index seek is different and returns different rows.

We refer to *CustomerId* as a correlated parameter. If a nested loops join has correlated parameters, they appear in the plan as OUTER REFERENCES. We often refer to this type of nested loops join in which we have an index seek that depends on a correlated parameter as an index join. An index join is possibly the most common type of nested loops join. In fact, in SQL Server 2005, as we've already seen, a bookmark lookup is simply an index join between a nonclustered index and the base table.

The prior example illustrated two important techniques that SQL Server uses to boost the performance of a nested loops join: correlated parameters and, more importantly, an index seek based on those correlated parameters on the inner side of the join. Another performance optimization that we don't see here is the use of a lazy spool on the inner side of the join. A lazy spool caches and can reaccess the results from the inner side of the join. A lazy spool is especially useful when there are correlated parameters with many duplicate values and when the inner side of the join is relatively expensive to evaluate. By using a lazy spool, SQL Server can avoid recomputing the inner side of the join multiple times with the same correlated parameters. We will see some examples of spools including lazy spools later in this chapter.

Not all nested loops joins have correlated parameters. A simple way to get a nested loops join without correlated parameters is with a cross join. A cross join matches all rows of one table with all rows of the other table. To implement a cross join with a nested loops join, we must scan and join every row of the inner table to every row of the outer table. The set of inner table rows does not change depending on which outer table row we are processing. Thus, with a cross join, there can be no correlated parameter.

In some cases, if we do not have a suitable index or if we do not have a join predicate that is suitable for an index seek, the optimizer may generate a query plan without correlated parameters. The rules for determining whether a join predicate is suitable for use with an index seek are identical to the rules for determining whether any other predicate is suitable for an index seek. For example, consider the following query, which returns the number of employees who were hired after each other employee:

```
SELECT E1.[EmployeeId], COUNT(*)
FROM [Employees] E1 JOIN [Employees] E2
   ON E1.[HireDate] < E2.[HireDate]
GROUP BY E1.[EmployeeId]
```

We have no index on the *HireDate* column. Thus, this query generates a simple nested loops join with a predicate but without any correlated parameters and without an index seek:

```
|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1007],0)))
     |--Stream Aggregate(GROUP BY:([E1].[EmployeeID]) DEFINE:([Expr1007]=Count(*)))
          |--Nested Loops(Inner Join, WHERE:([E1].[HireDate]<[E2].[HireDate]))
             |--Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E1]))
             |--Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E2]))
```

Do not be concerned with the aggregation. The purpose of this example is to illustrate the behavior of the nested loops join. We will discuss aggregation later in this chapter.

Now consider the following identical query which has been rewritten to use a CROSS APPLY:

```
SELECT E1.[EmployeeId], ECnt.Cnt
FROM [Employees] E1 CROSS APPLY
(
   SELECT COUNT(*) Cnt
   FROM [Employees] E2
   WHERE E1.[HireDate] < E2.[HireDate]
) ECnt
```

Although these two queries are identical, and will always return the same results, the plan for the query with the CROSS APPLY uses a nested loops join with a correlated parameter:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([E1].[HireDate]))
     |--Clustered Index Scan(OBJECT:([Employees].[PK_Employees] AS [E1]))
     |--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1007],0)))
          |--Stream Aggregate(DEFINE:([Expr1007]=Count(*)))
               |--Clustered Index Scan (OBJECT:([Employees].[PK_Employees] AS [E2]),
                    WHERE:([E1].[HireDate]<[E2].[HireDate]))
```

Tip This example demonstrates that, in some cases, small changes to a query, even rewrites that do not change the semantics of the query, can yield substantially different query plans. In particular, in some cases, a CROSS APPLY can induce the optimizer to generate a correlated nested loops join, which may or may not be desirable.

Join Predicates and Logical Join Types The nested loops join is one of the most flexible join methods. It supports all join predicates including equijoin (equality) predicates and inequality predicates.

The nested loops join supports the following logical join operators:

- Inner join
- Left outer join
- Cross join
- Cross apply and outer apply
- Left semi-join and left anti-semi-join

The nested loops join does not support the following logical join operators:

- Right and full outer join
- Right semi-join and right anti-semi-join

To understand why the nested loops join does not support right outer or semi-joins, let's first look at how we can extend the nested loops join algorithm to support left outer and semi-joins. Here is the pseudo-code for a left outer join. We need only two extra lines of pseudo-code to extend the inner join algorithm.

```
for each row R1 in the outer table
   begin
      for each row R2 in the inner table
         if R1 joins with R2
```

```
                output (R1, R2)
            **if R1 did not join**
                **output (R1, NULL)**
    end
```

This algorithm keeps track of whether we joined a particular outer row. If after exhausting all inner rows, we find that a particular inner row did not join, we output it as a NULL extended row. We can write similar pseudo-code for a left semi-join or left anti-semi-join. [A semi-join or anti-semi-join returns one half of the input information, that is, columns from one of the joined tables. So instead of outputting (R1, R2) as in the pseudo-code above, a left semi-join outputs just R1. Moreover, a semi-join returns each row of the outer table at most once. Thus, after finding a match and outputting a given row R1, a left semi-join moves immediately to the next outer row. A left anti-semi-join returns a row from R1 if it does *not* match with R2.]

Now consider how we might support right outer join. In this case, we want to return pairs (R1, R2) for rows that join and pairs (NULL, R2) for rows of the inner table that do not join. The problem is that we scan the inner table multiple times—once for each row of the outer join. We may encounter the same inner rows multiple times during these multiple scans. At what point can we conclude that a particular inner row has not or will not join? Moreover, if we are using an index join, we might not encounter some inner rows at all. Yet these rows should also be returned for an outer join. Further analysis uncovers similar problems for right semi-joins and right anti-semi-joins.

Fortunately, since right outer join commutes into left outer join and right semi-join commutes into left semi-join, SQL Server can use the nested loops join for right outer and semi-joins. However, while these transformations are valid, they may affect performance. When the optimizer transforms a right join into a left join, it also switches the outer and inner inputs to the join. Recall that to use an index join, the index needs to be on the inner table. By switching the outer and inner inputs to the table, the optimizer also switches the table on which we need an index to be able to use an index join.

Full Outer Joins The nested loops join cannot directly support full outer join. However, the optimizer can transform [Table1] FULL OUTER JOIN [Table2] into [Table1] LEFT OUTER JOIN [Table2] UNION ALL [Table2] LEFT ANTI-SEMI-JOIN [Table1]. Basically, this transforms the full outer join into a left outer join—which includes all pairs of rows from *Table1* and *Table2* that join and all rows of *Table1* that do not join—then adds back the rows of *Table2* that do not join using an anti-semi-join. To demonstrate this transformation, suppose that we have two customer tables. Further suppose that each customer table has different customer ids. We want to merge the two lists while keeping track of the customer ids from each table. We want the result to include all customers regardless of whether a customer appears in both lists or in just one list. We can generate this result with a full outer join. We'll make the rather unrealistic assumption that two customers with the same name are indeed the same customer.

```
CREATE TABLE [Customer1] ([CustomerId] int PRIMARY KEY, [Name] nvarchar(30))
CREATE TABLE [Customer2] ([CustomerId] int PRIMARY KEY, [Name] nvarchar(30))

SELECT C1.[Name], C1.[CustomerId], C2.[CustomerId]
FROM [Customer1] C1 FULL OUTER JOIN [Customer2] C2
     ON C1.[Name] = C2.[Name]
```

Here is the plan for this query, which demonstrates the transformation in action:

```
|--Concatenation
     |--Nested Loops(Left Outer Join, WHERE:([C1].[Name]=[C2].[Name]))
     |    |--Clustered Index Scan(OBJECT:([Customer1].[PK_Customer1] AS [C1]))
     |    |--Clustered Index Scan(OBJECT:([Customer2].[PK_Customer2] AS [C2]))
     |--Compute Scalar(DEFINE:([C1].[CustomerId]=NULL, [C1].[Name]=NULL))
          |--Nested Loops(Left Anti Semi Join, WHERE:([C1].[Name]=[C2].[Name]))
             |--Clustered Index Scan(OBJECT:([Customer2].[PK_Customer2] AS [C2]))
             |--Clustered Index Scan(OBJECT:([Customer1].[PK_Customer1] AS [C1]))
```

The concatenation operator implements the UNION ALL. We'll cover this operator in a bit more detail when we discuss unions later in this chapter.

Costing The complexity or cost of a nested loops join is proportional to the size of the outer input multiplied by the size of the inner input. Thus, a nested loops join generally performs best for relatively small input sets. The inner input need not be small, but, if it is large, it helps to include an index on a highly selective join key.

In some cases, a nested loops join is the only join algorithm that SQL Server can use. SQL Server must use a nested loops join for cross join as well as for some complex cross applies and outer applies. Moreover, as we are about to see, with one exception, a nested loops join is the only join algorithm that SQL Server can use without at least one equijoin predicate. In these cases, the optimizer must choose a nested loops join regardless of cost.

Note Merge join supports full outer joins without an equijoin predicate. We will discuss this unusual scenario in the next section.

Partitioned Tables In SQL Server 2005, the nested loops join is also used to implement query plans that scan partitioned tables. To see an example of this use of the nested loops join, we need to create a partitioned table. The following script creates a simple partition function and scheme that defines four partitions, creates a partitioned table using this scheme, and then selects rows from the table:

```
CREATE PARTITION FUNCTION [PtnFn] (int) AS RANGE FOR VALUES (1, 10, 100)
CREATE PARTITION SCHEME [PtnSch] AS PARTITION [PtnFn] ALL TO ([PRIMARY])
CREATE TABLE [PtnTable] ([PK] int PRIMARY KEY, [Data] int) ON [PtnSch]([PK])

SELECT [PK], [Data] FROM [PtnTable]
```

SQL Server assigns sequential partition ids to each of the four partitions defined by the partition scheme. The range for each partition is shown in Table 3-4.

Table 3-4 The Range of Values in Each of Our Four Partitions

PartitionID	Values
1	[PK] <= 1
2	1 < [PK] <= 10
3	10 < [PK] <= 100
4	100 < [PK]

The query plan for the SELECT statement uses a constant scan operator to enumerate these four partition ids and a special nested loops join to execute a clustered index scan of each of these four partitions:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([PtnIds1003]) PARTITION ID:([PtnIds1003]))
   |--Constant Scan(VALUES:(((1)),((2)),((3)),((4))))
   |--Clustered Index Scan(OBJECT:([PtnTable].[PK__PtnTable]))
```

Observe that the nested loops join explicitly identifies the *partition id* column as *[PtnIds1003]*. Although it is not obvious from the text plan, the clustered index scan uses the *partition id* column and checks it on each execution to determine which partition to scan. This information is clearly visible in XML plans:

```
<RelOp NodeId="6" PhysicalOp="Clustered Index Scan" LogicalOp="Clustered Index Scan"...>
  <IndexScan Ordered="0" ForcedIndex="0" NoExpandHint="0">
     <Object... Table="[PtnTable]" Index="[PK_PtnTable]" />
       <PartitionId>
          <ColumnReference Column="PtnIds1003" />
       </PartitionId>
  </IndexScan>
</RelOp>
```

Merge Join

Now let's look at merge join. Unlike the nested loops join, which supports any join predicate, the merge join requires at least one equijoin predicate. Moreover, the inputs to the merge join must be sorted on the join keys. For example, if we have a join predicate [Customers].[CustomerId] = [Orders].[CustomerId], the *Customers* and *Orders* tables must both be sorted on the *CustomerId* column.

The merge join works by simultaneously reading and comparing the two sorted inputs one row at a time. At each step, it compares the next row from each input. If the rows are equal, it outputs a joined row and continues. If the rows are not equal, it discards the lesser of the two inputs and continues. Since the inputs are sorted, any row that the join discards must be less than any of the remaining rows in either input and, thus, can never join. A merge join does not necessarily need to scan every row from both inputs. As soon as it reaches the end of either input, the merge join stops scanning.

We can express the algorithm in pseudo-code as:

```
get first row R1 from input 1
get first row R2 from input 2
while not at the end of either input
   begin
      if R1 joins with R2
         begin
            output (R1, R2)
            get next row R2 from input 2
         end
      else if R1 < R2
         get next row R1 from input 1
      else
         get next row R2 from input 2
   end
```

Unlike the nested loops join where the total cost may be proportional to the *product* of the number of rows in the input tables, with a merge join each table is read at most once and the total cost is proportional to the *sum* of the number of rows in the inputs. Thus, merge join is often a better choice for larger inputs.

One-to-Many vs. Many-to-Many Merge Join The above pseudo-code implements a one-to-many merge join. After it joins two rows, it discards R2 and moves to the next row of input 2. This presumes that it will never find another row from input 1 that will ever join with the discarded row. In other words, there can't be duplicates in input 1. On the other hand, it is acceptable that there might be duplicates in input 2 since it did not discard the current row from input 1.

Merge join can also support many-to-many merge joins. In this case, it must keep a copy of each row from input 2 whenever it joins two rows. This way, if it later finds a duplicate row from input 1, it can play back the saved rows. On the other hand, if it finds that the next row from input 1 is not a duplicate, it can discard the saved rows. The merge join saves these rows in a worktable in *tempdb*. The amount of required disk space depends on the number of duplicates in input 2.

A one-to-many merge join is always more efficient than a many-to-many merge join since it does not need a worktable. To use a one-to-many merge join, the optimizer must be able to determine that one of the inputs consists strictly of unique rows. Typically, this means that either there is a unique index on the input or there is an explicit operator in the plan (perhaps a sort distinct or a group by) to ensure that the input rows are unique.

Sort Merge Join vs. Index Merge Join There are two ways that SQL Server can get sorted inputs for a merge join: It may explicitly sort the inputs using a sort operator, or it may read the rows from an index. In general, a plan using an index to achieve sort order is cheaper than a plan using an explicit sort.

Join Predicates and Logical Join Types Merge join supports multiple equijoin predicates as long as the inputs are sorted on all of the join keys. The specific sort order does not matter as long as both inputs are sorted in the same order. For example, if we have a join predicate T1.[Col1] = T2.[Col1] and T1.[Col2] = T2.[Col2], we can use a merge join as long as tables *T1* and *T2* are both sorted either on (Col1, Col2) or on (Col2, Col1).

Merge join also supports residual predicates. For example, consider the join predicate T1.[Col1] = T2.[Col1] and T1.[Col2] > T2.[Col2]. Although the inequality predicate cannot be used as part of a merge join, the equijoin portion of this predicate can be used to perform a merge join (presuming both tables are sorted on [Col1]). For each pair of rows that joins on the equality portion of predicate, the merge join can then apply the inequality predicate. If the inequality evaluates to true, the join returns the row; if not, it discards the row.

Merge join supports all outer and semi-join variations. For instance, to implement an outer join, the merge join simply needs to track whether each row has joined. Instead of discarding a row that has not joined, it can NULL extend it and output it as appropriate. Note that, unlike the inner join case where a merge join can stop as soon as it reaches the end of either input, for an outer (or anti-semi-) join the merge join must scan to the end of whichever input it is preserving. For a full outer join, it must scan to the end of both inputs.

Merge join supports a special case for full outer join. In some cases, the optimizer generates a merge join for a full outer join even if there is no equijoin predicate. This join is equivalent to a many-to-many merge join where all rows from one input join with all rows from the other input. As with any other many-to-many merge join, SQL Server builds a worktable to store and play back all rows from the second input. SQL Server supports this plan as an alternative to the previously discussed transformation used to support full outer join with nested loops join.

Examples Because merge join requires that input rows be sorted, the optimizer is most likely to choose a merge join when we have an index that returns rows in that sort order. For example, the following query simply joins the *Orders* and *Customers* tables:

```
SELECT O.[OrderId], C.[CustomerId], C.[ContactName]
FROM [Orders] O JOIN [Customers] C
   ON O.[CustomerId] = C.[CustomerId]
```

Since we have no predicates other than the join predicates, we must scan both tables in their entirety. Moreover, we have covering indexes on the *CustomerId* column of both tables. Thus, the optimizer chooses a merge join plan:

```
|--Merge Join(Inner Join, MERGE:([C].[CustomerID])=([O].[CustomerID]), RESIDUAL:(...))
   |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]), ORDERED FORWARD)
   |--Index Scan(OBJECT:([Orders].[CustomerID] AS [O]), ORDERED FORWARD)
```

Observe that this join is one to many. We can tell that it is one to many by the absence of the MANY-TO-MANY keyword in the query plan. We have a unique index (actually a primary key) on the *CustomerId* column of the *Customers* table. Thus, the optimizer knows that there will be no duplicate *CustomerId* values from this table and chooses the one-to-many join.

Note that for a unique index to enable a one-to-many join, we must be joining on all of the key columns of the unique index. It is not sufficient to join on a subset of the key columns as the index only guarantees uniqueness on the entire set of key columns.

Now let's consider a slightly more complex example. The following query returns a list of orders that shipped to cities different from the city that we have on file for the customer who placed the order:

```
SELECT O.[OrderId], C.[CustomerId], C.[ContactName]
FROM [Orders] O JOIN [Customers] C
   ON O.[CustomerId] = C.[CustomerId] AND O.[ShipCity] <> C.[City]
ORDER BY C.[CustomerId]
```

We need the ORDER BY clause to encourage the optimizer to choose a merge join. We'll return to this point in a moment. Here is the query plan:

```
|--Merge Join(Inner Join, MERGE:([C].[CustomerID])=([O].[CustomerID]),
   RESIDUAL:(... AND [O].[ShipCity]<>[C].[City]))
     |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]), ORDERED FORWARD)
     |--Sort(ORDER BY:([O].[CustomerID] ASC))
          |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O]))
```

There are a couple of points worth noting about this new plan. First, because this query needs the *ShipCity* column from the *Orders* table for the extra predicate, the optimizer cannot use a scan of the *CustomerId* index, which does not cover the extra column, to get rows from the *Orders* table sorted by the *CustomerId* column. Instead, the optimizer chooses to scan the clustered index and sort the results. The ORDER BY clause requires that the optimizer add this sort either before the join, as in this example, or after the join. By performing the sort before the join, the plan can take advantage of the merge join. Moreover, the merge join preserves the input order so there is no need to sort the data again after the join.

Note Technically, the optimizer could decide to use a scan of the *CustomerId* index along with a bookmark lookup, but since it is scanning the entire table, the bookmark lookup would be prohibitively expensive.

Second, this merge join demonstrates a residual predicate: *O.[ShipCity] <> C.[City]*. The optimizer cannot use this predicate as part of the join's merge keys because it is an inequality. However, as the example shows, as long as there is at least one equality predicate, SQL Server can use the merge join.

Hash Join

Hash join is the third physical join operator. When it comes to physical join operators, hash join does the heavy lifting. While nested loops join works well with relatively small data sets and merge join helps with moderately-sized data sets, hash join excels at performing the largest joins. Hash joins parallelize and scale better than any other join and are great at minimizing response times for data warehouse queries.

Hash join shares many characteristics with merge join. Like merge join, it requires at least one equijoin predicate, supports residual predicates, and supports all outer and semi-joins. Unlike merge join, it does not require ordered input sets and, while it does support full outer join, it does require an equijoin predicate.

The hash join algorithm executes in two phases known as the "build" and "probe" phases. During the build phase, it reads all rows from the first input (often called the left or build input), hashes the rows on the equijoin keys, and creates or builds an in-memory hash table. During the probe phase, it reads all rows from the second input (often called the right or probe input), hashes these rows on the same equijoin keys, and looks or probes for matching rows in the hash table. Since hash functions can lead to collisions (two different key values that hash to the same value), the hash join typically must check each potential match to ensure that it really joins. Here is pseudo-code for this algorithm:

```
for each row R1 in the build table
    begin
        calculate hash value on R1 join key(s)
        insert R1 into the appropriate hash bucket
    end
for each row R2 in the probe table
    begin
        calculate hash value on R2 join key(s)
        for each row R1 in the corresponding hash bucket
            if R1 joins with R2
                output (R1, R2)
    end
```

Note that unlike the nested loops and merge joins, which immediately begin flowing output rows, the hash join is blocking on its build input. That is, it must read and process its entire build input before it can return any rows. Moreover, unlike the other join methods, the hash join requires a memory grant to store the hash table. Thus, there is a limit to the number of concurrent hash joins that SQL Server can run at any given time. While these characteristics and restrictions are generally not a problem for data warehouses, they are undesirable for most OLTP applications.

Note A sort merge join does require a memory grant for the sort operator(s) but does not require a memory grant for the merge join itself.

Memory and Spilling Before a hash join begins execution, SQL Server tries to estimate how much memory it will need to build its hash table. It uses the cardinality estimate for the size of the build input along with the expected average row size to estimate the memory requirement. To minimize the memory required by the hash join, the optimizer chooses the smaller of the two tables as the build table. SQL Server then tries to reserve sufficient memory to ensure that the hash join can successfully store the entire build table in memory.

What happens if SQL Server grants the hash join less memory than it requests or if the estimate is too low? In these cases, the hash join may run out of memory during the build phase. If the hash join runs out of memory, it begins spilling a small percentage of the total hash table to disk (to a workfile in *tempdb*). The hash join keeps track of which buckets of the hash table are still in memory and which ones have been spilled to disk. As it reads each new row from the build table, it checks to see whether it hashes to an in-memory or an on-disk bucket. If it hashes to an in-memory bucket, it proceeds normally. If it hashes to an on-disk bucket, it writes the row to disk. This process of running out of memory and spilling buckets to disk may repeat multiple times until the build phase is complete.

The hash join performs a similar process during the probe phase. For each new row from the probe table, it checks to see whether it hashes to an in-memory or an on-disk bucket. If it hashes to an in-memory bucket, it probes the hash table, produces any appropriate joined rows, and discards the row. If it hashes to an on-disk bucket, it writes the row to disk. Once the join completes the first pass of the probe table, it returns one by one to any buckets that spilled, reads the build rows back into memory, reconstructs the hash table for each bucket, and then reads the corresponding probe bucket and completes the join. If while processing spilled sets of buckets, the hash join again runs out of memory, the process simply repeats. We refer to the number of times that the hash join repeats this algorithm and spills the same data as the "recursion level." After a set number of recursion levels, if the hash join continues to spill, it switches to a special "bailout" algorithm that, while less efficient, is guaranteed to complete eventually.

Left Deep vs. Right Deep vs. Bushy Hash Join Trees The shape and order of joins in a query plan can significantly impact the performance of the plan. The shape of a query plan is so important that we actually have terms for the most common shapes. The terms—*left deep*, *right deep,* and *bushy*—are based on the physical appearance of the query plan, as illustrated by Figure 3-11.

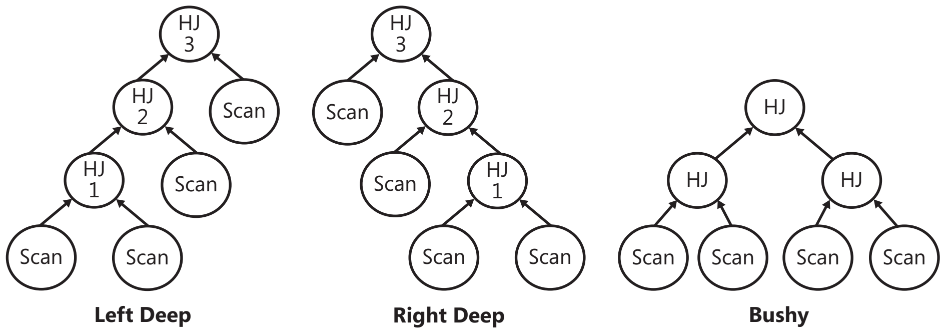


Figure 3-11 Three common shapes for query plans involving joins

The shape of the join tree is particularly interesting for hash joins as it affects the memory consumption.

In a left deep tree, the output of one hash join is the build input to the next hash join. Because hash joins consume their entire build input before moving to the probe phase, in a left deep tree only adjacent pairs of hash joins are active at the same time. For example, for the left deep example in Figure 3-11, SQL Server begins by building the hash table for HJ1. When HJ1 begins probing, HJ2 begins building its hash table. When HJ1 is done probing, SQL Server can release the memory used by its hash table. Only then does HJ2 begin probing and HJ3 begin building its hash table. Thus, HJ1 and HJ3 are never active at the same time and can share the same memory grant. The total memory requirement is the maximum of the memory needed by any two adjacent joins (that is, HJ1 and HJ2 or HJ2 and HJ3).

In a right deep tree, the output of one hash join is the probe input to the next hash join. All of the hash joins build their complete hash tables before any begin the probe phase of the join. All of the hash joins are active at once and cannot share memory. When SQL Server does begin the probe phase of the join, the rows flow up the entire tree of hash joins without blocking. Thus, the total memory requirement is the sum of the memory needed by all three joins.

Examples The following query is nearly identical to the earlier merge join example except that we select one additional column, the *OrderDate* column, from the *Orders* table:

```
SELECT O.[OrderId], O.[OrderDate], C.[CustomerId], C.[ContactName]
FROM [Orders] O JOIN [Customers] C
   ON O.[CustomerId] = C.[CustomerId]
```

Because the *CustomerId* index on the *Orders* table does not cover the *OrderDate* column, we would need a sort to use a merge join. We saw this outcome in the second merge join example, but this time we do not have an ORDER BY clause. Thus, the optimizer chooses the following hash join plan:

```
|--Hash Match(Inner Join, HASH:([C].[CustomerID])=([O].[CustomerID]), RESIDUAL:(...))
   |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]))
   |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O]))
```

Summary of Join Properties

Table 3-5 summarizes the characteristics of the three physical join operators.

Table 3-5 Characteristics of the Three Join Algorithms

	Nested Loops Join	Merge Join	Hash Join
Best for . . .	Relatively small inputs with an index on the inner table on the join key.	Medium to large inputs with indexes to provide order on the equijoin keys and/or where we require order after the join.	Data warehouse queries with medium to large inputs. Scalable parallel execution.
Concurrency	Supports large numbers of concurrent users.	Many-to-one join with order provided by indexes (rather than explicit sorts) supports large numbers of concurrent users.	Best for small numbers of concurrent users.
Stop and go	No	No	Yes (build input only)
Equijoin required	No	Yes (except for full outer join)	Yes
Outer and semi-joins	Left joins only (full outer joins via transformation)	All join types	All join types
Uses memory	No	No (may require sorts which use memory)	Yes
Uses *tempdb*	No	Yes (many-to-many join only)	Yes (if join runs out of memory and spills)
Requires order	No	Yes	No
Preserves order	Yes (outer input only)	Yes	No
Supports dynamic cursors	Yes	No	No

Aggregations

SQL Server supports two physical operators for performing aggregations. These operators are stream aggregate and hash aggregate.

Scalar Aggregation

Scalar aggregates are queries with aggregate functions in the select list and no GROUP BY clause. Scalar aggregates always return a single row. SQL Server always implements scalar aggregates using the stream aggregate operator.

Let's begin by considering a trivial example:

```
SELECT COUNT(*) FROM [Orders]
```

This query produces the following plan:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1004],0)))
     |--Stream Aggregate(DEFINE:([Expr1004]=Count(*)))
          |--Index Scan(OBJECT:([Orders].[ShippersOrders]))
```

The stream aggregate operator just counts the number of input rows and returns this result. The stream aggregate actually computes the count ([Expr1004]) as a BIGINT. The compute scalar is needed to convert this result to the expected output type of INT. Note that a scalar stream aggregate is one of the only examples of a nonleaf operator that can produce an output row even with an empty input set.

It is easy to see how to implement other simple scalar aggregate functions such as MIN, MAX, and SUM. A single-stream aggregate operator can calculate multiple scalar aggregates at the same time:

```
SELECT MIN([OrderDate]), MAX([OrderDate]) FROM [Orders]
```

Here is the query plan with a single-stream aggregate operator:

```
|--Stream Aggregate(DEFINE:([Expr1003]=MIN([Orders].[OrderDate]),
   [Expr1004]=MAX([Orders].[OrderDate])))
     |--Index Scan(OBJECT:([Orders].[OrderDate]))
```

Note that SQL Server does not need to convert the result for the MIN and MAX aggregates since the data types of these aggregates are computed based on the data type of the *OrderDate* column.

Some aggregates such as AVG are actually calculated from two other aggregates such as SUM and COUNT:

```
SELECT AVG([Freight]) FROM [Orders]
```

Notice how the compute scalar operator in the plan computes the average from the sum and count:

```
|--Compute Scalar(DEFINE:([Expr1003]=CASE WHEN [Expr1004]=(0)
   THEN NULL
   ELSE [Expr1005]/CONVERT_IMPLICIT(money,[Expr1004],0)
   END))
   |--Stream Aggregate(DEFINE:([Expr1004]=COUNT_BIG([Orders].[Freight]),
      Expr1005]=SUM([Orders].[Freight])))
          |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

The CASE expression is needed to make sure that SQL Server does not attempt to divide by zero.

Although SUM does not need to be computed per se, it still needs the count:

```
SELECT SUM([Freight]) FROM [Orders]
```

Notice how the CASE expression in this query plan uses the COUNT to ensure that SUM returns NULL instead of zero if there are no rows:

```
|--Compute Scalar(DEFINE:([Expr1003]=CASE WHEN [Expr1004]=(0)
   THEN NULL
   ELSE [Expr1005]
   END))
   |--Stream Aggregate(DEFINE:([Expr1004]=COUNT_BIG([Orders].[Freight]),
      [Expr1005]=SUM([Orders].[Freight])))
        |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

Scalar Distinct Now let's take a look at what happens if we add a DISTINCT keyword to a scalar aggregate. Consider this query to compute the number of distinct cities to which we've shipped orders:

```
SELECT COUNT(DISTINCT [ShipCity]) FROM [Orders]
```

This query produces this query plan:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
     |--Stream Aggregate(DEFINE:([Expr1006]=COUNT([Orders].[ShipCity])))
          |--Sort(DISTINCT ORDER BY:([Orders].[ShipCity] ASC))
               |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

Since the query must only count rows that have a unique value for the *ShipCity* column, SQL Server adds a sort distinct operator to eliminate rows with duplicate *ShipCity* values. Sort distinct is one of the common methods used by SQL Server to eliminate duplicates. It is easy to remove duplicate rows after sorting the input set since the duplicates are then adjacent to one another. There are other methods that SQL Server can employ to eliminate duplicates, as we'll see shortly. Other than the addition of the sort operator, this plan is the same as the COUNT(*) plan with which we began our discussion of aggregation.

Not all distinct aggregates require duplicate elimination. For example, MIN and MAX behave identically with and without the distinct keyword. The minimum and maximum values of a set remain the same whether or not the set includes duplicate values. For example, this query gets the same plan as the above MIN/MAX query without the DISTINCT keyword.

```
SELECT MIN(DISTINCT [OrderDate]), MAX(DISTINCT [OrderDate]) FROM [Orders]
```

If we have a unique index, SQL Server also can skip the duplicate elimination because the index guarantees that there are no duplicates. For example, the following query is identical to the simple COUNT(*) query with which we began this discussion:

```
SELECT COUNT(DISTINCT [OrderId]) FROM [Orders]
```

Multiple Distinct Consider this query:

```
SELECT COUNT(DISTINCT [ShipAddress]), COUNT(DISTINCT [ShipCity])
FROM [Orders]
```

As we've seen, SQL Server can compute *COUNT(DISTINCT [ShipAddress])* by eliminating rows that have duplicate values for the *ShipAddress* column. Similarly, SQL Server can compute *COUNT(DISTINCT [ShipCity])* by eliminating rows that have duplicate values for the *ShipCity* column. But, given that these two sets of rows are different, how can SQL Server compute both at the same time? The answer is it cannot. It must first compute one aggregate result, then the other, and then it must combine the two results into a single output row:

```
|--Nested Loops(Inner Join)
     |--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1009],0)))
     |   |--Stream Aggregate(DEFINE:([Expr1009]=COUNT([Orders].[ShipAddress])))
     |      |--Sort(DISTINCT ORDER BY:([Orders].[ShipAddress] ASC))
     |         |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
     |--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1010],0)))
        |--Stream Aggregate(DEFINE:([Expr1010]=COUNT([Orders].[ShipCity])))
           |--Sort(DISTINCT ORDER BY:([Orders].[ShipCity] ASC))
              |--Clustered Index Scan
                 (OBJECT:([Orders].[PK_Orders]))
```

The two inputs to the nested loops join compute the two counts from the original query. One of the inputs removes duplicates and computes the count for the *ShipAddress* column, while the other input removes duplicates and computes the count for the *ShipCity* column. The nested loops join has no join predicate; it is a cross join. Since both inputs to the nested loops join each produce a single row—they are both scalar aggregates—the result of the cross join is also a single row. The cross join just serves to "glue" the two columns of the result into a single row.

If we have more than two distinct aggregates on different columns, the optimizer just uses more than one cross join. The optimizer also uses this type of plan for a mix of nondistinct and distinct aggregates. In that case, the optimizer uses a single cross join input to calculate all of the nondistinct aggregates.

Stream Aggregation

Now that we've seen how to compute scalar aggregates, let's see how SQL Server computes general purpose aggregates where we have a GROUP BY clause. We'll begin by taking a closer look at how stream aggregation works.

The Algorithm Stream aggregate relies on data arriving sorted by the GROUP BY column(s). Like a merge join, if a query includes a GROUP BY clause with more than one column, the stream aggregate can use any sort order that includes all of the columns. For example, a stream aggregate can group on columns *Col1* and *Col2* with data sorted on (*Col1, Col2*) or on (*Col2, Col1*). As with merge join, the sort order may be delivered by an index or by an explicit sort operator. The sort order ensures that sets of rows with the same value for the GROUP BY columns will be adjacent to one another.

Here is pseudo-code for the stream aggregate algorithm:

```
clear the current aggregate results
clear the current group by columns
for each input row
   begin
      if the input row does not match the current group by columns
         begin
            output the current aggregate results (if any)
            clear the current aggregate results
            set the current group by columns to the input row
         end
      update the aggregate results with the input row
   end
```

For example, to compute a SUM, the stream aggregate considers each input row. If the input row belongs to the current group (that is, the *group by* columns of the input row match the *group by* columns of the previous row), the stream aggregate updates the current SUM by adding the appropriate value from the input row to the running total. If the input row belongs to a new group (that is, the *group by* columns of the input row do not match the *group by* columns of the previous row), the stream aggregate outputs the current SUM, resets the SUM to zero, and starts a new group.

Simple Examples Consider the following query that counts the number of orders shipped to each address:

```
SELECT [ShipAddress], [ShipCity], COUNT(*)
FROM [Orders]
GROUP BY [ShipAddress], [ShipCity]
```

Here is the plan for this query:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
     |--Stream Aggregate(GROUP BY: ([Orders].[ShipCity], [Orders].[ShipAddress])
        DEFINE:([Expr1006]=Count(*)))
        |--Sort(ORDER BY:([Orders].[ShipCity] ASC, [Orders].[ShipAddress] ASC))
        |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

This plan is basically the same as the one that we saw for the scalar aggregate queries, except that SQL Server needs to sort the data before it aggregates. We can think of the scalar aggregate as one big group containing all of the rows; thus, for a scalar aggregate there is no need to sort the rows into different groups.

Stream aggregate preserves the input sort order. For example, suppose that we extended the above query with an ORDER BY clause that sorts the results on the GROUP BY keys:

```
SELECT [ShipAddress], [ShipCity], COUNT(*)
FROM [Orders]
GROUP BY [ShipAddress], [ShipCity]
ORDER BY [ShipAddress], [ShipCity]
```

The resulting plan still has only the one sort operator (below the stream aggregate):

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
      |--Stream Aggregate(GROUP BY:([Orders].[ShipAddress],[Orders].[ShipCity])
         DEFINE:([Expr1006]=Count(*)))
         |--Sort(ORDER BY:([Orders].[ShipAddress] ASC, [Orders].[ShipCity] ASC))
               |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

However, note that the sort columns are reversed from the first example. Previously, it did not matter whether SQL Server sorted on the *ShipAddress* or *ShipCity* column first. Now since the query includes an ORDER BY clause, the optimizer chooses to sort on the *ShipAddress* column first to avoid needing a second sort operator following the stream aggregate.

If we have an appropriate index, the plan does not need a sort operator at all. For instance, consider the following query which counts orders by customer (instead of shipping address):

```
SELECT [CustomerId], COUNT(*)
FROM [Orders]
GROUP BY [CustomerId]
```

The new plan uses an index on the *CustomerId* column to avoid sorting:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
   |--Stream Aggregate(GROUP BY:([Orders].[CustomerID]) DEFINE:([Expr1006]=Count(*)))
      |--Index Scan(OBJECT:([Orders].[CustomerID]), ORDERED FORWARD)
```

Select Distinct   If we have an index to provide order, SQL Server can also use the stream aggregate to implement SELECT DISTINCT. (If we do not have an index to provide order, the optimizer cannot use a stream aggregate without adding a sort. In this case, the optimizer can just let the sort distinct directly; there is no reason to use a stream aggregate as well.) SELECT DISTINCT is essentially the same as GROUP BY on all selected columns with no aggregate functions. For example:

```
SELECT DISTINCT [CustomerId] FROM [Orders]
```

Can also be written as:

```
SELECT [CustomerId] FROM [Orders] GROUP BY [CustomerId]
```

Both queries use the same plan:

```
|--Stream Aggregate(GROUP BY:([Orders].[CustomerID]))
   |--Index Scan(OBJECT:([Orders].[CustomerID]), ORDERED FORWARD)
```

Notice that the stream aggregate has a GROUP BY clause, but no defined columns.

Distinct Aggregates   SQL Server implements distinct aggregates for queries with a GROUP BY clause identically to how it implements distinct scalar aggregates. In both cases, a distinct aggregate needs a plan that eliminates duplicates before aggregating. For example, suppose we would like to find the number of distinct customers served by each employee:

```
SELECT [EmployeeId], COUNT(DISTINCT [CustomerId])
FROM [Orders]
GROUP BY [EmployeeId]
```

This query results in the following plan:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
       |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID])
          DEFINE:([Expr1006]=COUNT([Orders].[CustomerID])))
          |--Sort(DISTINCT ORDER BY:([Orders].[EmployeeID] ASC,[Orders].[CustomerID] ASC))
          |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

However, we just saw how SQL Server can use an aggregate to eliminate duplicates. SQL Server can also use an aggregate to implement distinct aggregates without the sort distinct. To see such a plan, we need to create a nonunique two-column index. For example, let's temporarily create an index on the *EmployeeId* and *CustomerId* columns of the *Orders* table:

```
CREATE INDEX [EmployeeCustomer] ON [Orders] (EmployeeId, CustomerId)
```

Now, the plan looks as follows:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
       |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID])
          DEFINE:([Expr1006]=COUNT([Orders].[CustomerID])))
           |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID], [Orders].[CustomerID]))
              |--Index Scan(OBJECT:([Orders].[EmployeeCustomer]), ORDERED FORWARD)
```

Observe how the optimizer has replaced the sort distinct with a stream aggregate. This plan is possible since, as in the SELECT DISTINCT example above, we have an index that provides order on the columns over which we need to eliminate duplicates. Let's drop the temporary index before we continue:

```
DROP INDEX [Orders].[EmployeeCustomer]
```

Multiple Distincts Finally, let's take a look at how SQL Server implements a mix of nondistinct and distinct aggregates (or multiple distinct aggregates) in a single GROUP BY query. Suppose that we wish to find the total number of orders taken by each employee, as well as the total number of distinct customers served by each employee:

```
SELECT [EmployeeId], COUNT(*), COUNT(DISTINCT [CustomerId])
FROM [Orders]
GROUP BY [EmployeeId]
```

As in the scalar aggregate with multiple distincts example that we saw earlier, SQL Server cannot compute the two aggregates at the same time. Instead, it must compute each aggregate separately. However, unlike the scalar aggregate example, the GROUP BY clause means each aggregate returns multiple rows. Thus, while the scalar aggregate plan used a cross join to glue together two individual rows, this query needs an inner join on *EmployeeId* (the *group by* column) to combine the two sets of rows. Here is the query plan:

```
|--Compute Scalar(DEFINE:([Orders].[EmployeeID]=[Orders].[EmployeeID]))
   |--Merge Join(Inner Join, MANY-TO-MANY
      MERGE:([Orders].[EmployeeID])=([Orders].[EmployeeID]),RESIDUAL:(...))
       |--Compute Scalar(DEFINE:([Orders].[EmployeeID]=[Orders].[EmployeeID]))
       |    |--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1011],0)))
       |       |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID])
                  DEFINE:([Expr1011]=COUNT([Orders].[CustomerID])))
       |          |--Sort(DISTINCT ORDER BY:([Orders].[EmployeeID] ASC,
                      [Orders].[CustomerID] ASC))
       |                |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
       |--Compute Scalar(DEFINE:([Orders].[EmployeeID]=[Orders].[EmployeeID]))
            |--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1012],0)))
               |--Stream Aggregate(GROUP BY:([Orders].[EmployeeID])
                  DEFINE:([Expr1012]=Count(*)))
                  |--Index Scan(OBJECT:([Orders].[EmployeeID]),ORDERED FORWARD)
```

Notice that since the results of the two aggregations are already sorted on the *group by* column, the plan can use a merge join. You can see the graphical plan for the query in Figure 3-12.

Note The compute scalar operators that appear to define *[EmployeeId] = [EmployeeId]* are needed for internal purposes and can be disregarded. The merge join ought to be one to many not many to many since the aggregates ensure uniqueness on the *group by* column (and *join* column). This is a minor performance issue not a correctness issue.

Hash Aggregation

The other aggregation operator, hash aggregate, is similar to hash join. It does not require (or preserve) sort order, requires memory, and is blocking (that is, it does not produce any results until it has consumed its entire input). Hash aggregate excels at efficiently aggregating very large data sets and, in parallel plans, scales better than stream aggregate.

Here is pseudo-code for the hash aggregate algorithm:

```
for each input row
   begin
      calculate hash value on group by column(s)
      check for a matching row in the hash table
      if matching row not found
         insert a new row into the hash table
      else
         update the matching row with the input row
   end
output all rows in the hash table
```

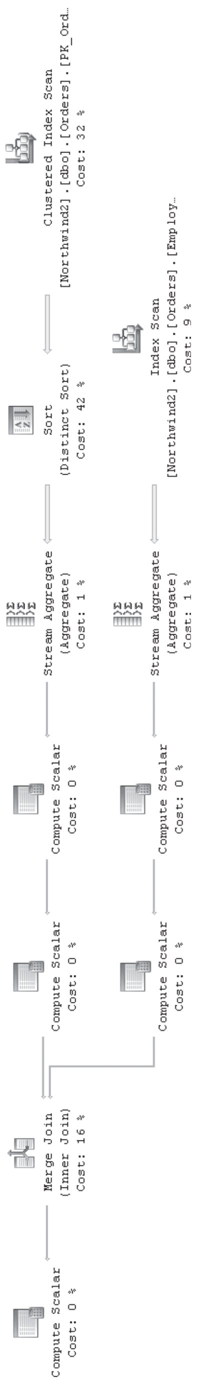


Figure 3-12 Graphical plan showing stream aggregation and a merge join to compute multiple distinct aggregates

While stream aggregate computes just one group at a time, hash aggregate computes all of the groups simultaneously. Like a hash join, a hash aggregate uses a hash table to store these groups. With each new input row, it checks the hash table to see whether the new row belongs to an existing group. If it does, it simply updates the existing group. If it does not, it creates a new group. Since the input data is unsorted, any row can belong to any group. Thus, a hash aggregate cannot output any results until it finishes processing every input row.

Memory and Spilling As with hash join, the hash aggregate requires memory. Before executing a query with a hash aggregate, SQL Server uses cardinality estimates to estimate how much memory it needs to execute the query. A hash join stores each build row, so the total memory requirement is proportional to the number and size of the build rows. The number of rows that join and the output cardinality of the join have no effect on the memory requirement of the join. A hash aggregate, on the other hand, stores only one row for each group, so the total memory requirement is actually proportional to the number and size of the output groups or rows. If there are fewer unique values of the *group by* column(s) and fewer groups, a hash aggregate needs less memory. If there are more unique values of the *group by* column(s) and more groups, a hash aggregate needs more memory.

Like hash join, if a hash aggregate runs out of memory, it must begin spilling rows to a workfile in *tempdb*. The hash aggregate spills one or more buckets, including any partially aggregated results along with any additional new rows that hash to the spilled buckets. Once the hash aggregate finishes processing all input rows, it outputs the completed in-memory groups and repeats the algorithm by reading back and aggregating one set of spilled buckets at a time. By dividing the spilled rows into multiple sets of buckets, the hash aggregate reduces the size of each set and, thus, reduces the risk that the algorithm will need to repeat many times.

Note that while duplicate rows are a potential problem for hash join, as they lead to skew in the size of the different hash buckets and make it difficult to divide the work into small uniform portions, duplicates can be quite helpful for hash aggregate since they collapse into a single group.

Examples The optimizer tends to favor hash aggregation for tables with more rows, fewer groups, no ORDER BY clause or other reason to sort, and no index that produces sorted rows. For example, our original stream aggregate example grouped the *Orders* table on the *ShipAddress* and *ShipCity* columns. This query produces 94 groups from 830 orders.
Now consider the following essentially identical query, which groups the *Orders* table on the *ShipCountry* column:

```
SELECT [ShipCountry], COUNT(*)
FROM [Orders]
GROUP BY [ShipCountry]
```

The *ShipCountry* column has only 21 unique values. Because a hash aggregate requires less memory as the number of groups decreases, whereas a sort requires memory proportional to the number of input rows, this time the optimizer chooses a plan with a hash aggregate:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
   |--Hash Match(Aggregate, HASH:([Orders].[ShipCountry]), RESIDUAL:(...)
      DEFINE:([Expr1006]=COUNT(*)))
      |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

Notice that the hash aggregate hashes on the *group by* column. (The residual predicate from the hash aggregate—which has been edited out of the above text plan to save space—is used to compare rows in the hash table to input rows in case of a hash value collision.) Also observe that with the hash aggregate no sort is needed. However, suppose we explicitly request a sort using an ORDER BY clause in the query:

```
SELECT [ShipCountry], COUNT(*)
FROM [Orders]
GROUP BY [ShipCountry]
ORDER BY [ShipCountry]
```

Because of the explicit ORDER BY clause, the plan must include a sort, thus the optimizer chooses a stream aggregate plan:

```
|--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
   |--Stream Aggregate(GROUP BY:([Orders].[ShipCountry]) DEFINE:([Expr1006]=Count(*)))
      |--Sort(ORDER BY:([Orders].[ShipCountry] ASC))
         |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

If the table gets big enough and the number of groups remains small enough, eventually the optimizer will decide that it is cheaper to use the hash aggregate and sort after the aggregation. For example, our *Northwind2* database includes a *BigOrders* table, which includes the same data from the original *Orders* table repeated five times. The *BigOrders* table has 4,150 rows compared to the 830 rows in the original *Orders* table. However, if we repeat the above query against the *BigOrders* table, we still get the same 21 groups:

```
SELECT [ShipCountry], COUNT(*)
FROM [BigOrders]
GROUP BY [ShipCountry]
ORDER BY [ShipCountry]
```

As the following plan shows, the optimizer concludes that it is better to use the hash aggregate and sort 21 rows than to sort 4,150 rows and use a stream aggregate:

```
|--Sort(ORDER BY:([BigOrders].[ShipCountry] ASC))
   |--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1007],0)))
      |--Hash Match(Aggregate, HASH:([BigOrders].[ShipCountry]),
         RESIDUAL:(...) DEFINE:([Expr1007]=COUNT(*)))
         |--Clustered Index Scan(OBJECT:([BigOrders].[OrderID]))
```

Figure 3-13 shows the graphical plan for this query. Note the sort is performed as the last operation, as the hash aggregation did not require that the data be sorted earlier.

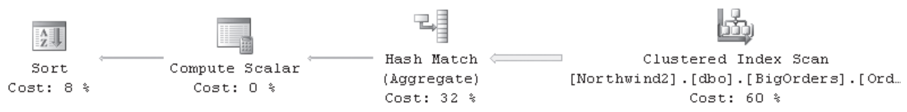


Figure 3-13 Graphical plan with hash aggregation and sorting

Distinct Just like stream aggregate, hash aggregate can be used to implement distinct operations. For example, suppose we just want a list of distinct countries to which we've shipped orders:

```
SELECT DISTINCT [ShipCountry] FROM [Orders]
```

Just as the optimizer chose a hash aggregate when we grouped on the *ShipCountry* column, it also chooses a hash aggregate for this query:

```
|--Hash Match(Aggregate, HASH:([Orders].[ShipCountry]), RESIDUAL:(...))
   |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
```

Finally, hash aggregate can be used to implement distinct aggregates, including multiple distincts. The basic idea is the same as for stream aggregate. SQL Server computes each aggregate separately and then joins the results together. For instance, suppose that, for each country to which we have shipped orders, we wish to find both the number of employees that have taken orders and the number of customers that have placed orders that were shipped to that country. For the optimizer to choose a plan with a hash aggregate, we need to run this query against a much larger table than we've used for our other examples. Our *Northwind2* database includes a *HugeOrders* table, which includes the same data from the original *Orders* table repeated 25 times.

```
SELECT [ShipCountry], COUNT(DISTINCT [EmployeeId]), COUNT(DISTINCT [CustomerId])
FROM [HugeOrders]
GROUP BY [ShipCountry]
```

The following query plan has several interesting features:

```
|--Compute Scalar(DEFINE:([HugeOrders].[ShipCountry]=[HugeOrders].[ShipCountry]))
   |--Hash Match(Inner Join, HASH:([HugeOrders].[ShipCountry])=
      ([HugeOrders].[ShipCountry]),RESIDUAL:(...))
      |--Compute Scalar(DEFINE:([HugeOrders].[ShipCountry]=[HugeOrders].[ShipCountry]))
      |     |--Compute Scalar(DEFINE:([Expr1005]=CONVERT_IMPLICIT(int,[Expr1012],0)))
      |        |--Hash Match(Aggregate, HASH:([HugeOrders].[ShipCountry]),RESIDUAL:(...)
                    DEFINE:([Expr1012]=COUNT([HugeOrders].[CustomerID])))
      |           |--Hash Match(Aggregate,HASH:([HugeOrders].[ShipCountry],
                       [HugeOrders].[CustomerID]),RESIDUAL:(...))
      |              |--Clustered Index Scan (OBJECT:([HugeOrders].[OrderID]))
      |--Compute Scalar(DEFINE:([HugeOrders].[ShipCountry]=
         [HugeOrders].[ShipCountry]))
```

```
    |--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(int,[Expr1013],0)))
       |--Stream Aggregate (GROUP BY:([HugeOrders].[ShipCountry])
          DEFINE:([Expr1013]=COUNT([HugeOrders].[EmployeeID])))
          |--Sort(ORDER BY:([HugeOrders].[ShipCountry]))
             |--Hash Match(Aggregate,HASH:([HugeOrders].[ShipCountry],
                [HugeOrders].[EmployeeID]),RESIDUAL:(...))
                |--Clustered Index Scan (OBJECT:([HugeOrders].[OrderID]))
```

First, this plan shows that SQL Server can mix hash aggregates and stream aggregates in a single plan. In fact, this plan uses both a hash aggregate and a stream aggregate in the computation of *COUNT(DISTINCT [EmployeeId])*. The hash aggregate eliminates duplicate values from the *EmployeeId* column, while the sort and stream aggregate compute the counts. Although SQL Server could use a sort distinct and eliminate the hash aggregate, the sort would take more memory.

Second, SQL Server uses a pair of hash aggregates to compute *COUNT(DISTINCT [CustomerId])*. This portion of the plan works exactly like the earlier distinct example that used two stream aggregates. The bottommost hash aggregate eliminates duplicate values from the *CustomerId* column, whereas the topmost computes the counts.

Third, because the hash aggregate does not return rows in any particular order, SQL Server cannot use a merge join without introducing another sort. Instead, the optimizer chooses a hash join for this plan. The graphical plan is shown in Figure 3-14.

Unions

There are two types of union queries: UNION ALL and UNION. A UNION ALL query simply combines the results from two or more different queries and returns the results. For example, here is a simple UNION ALL query that returns a list of all employees and customers:

```
SELECT [FirstName] + N' ' + [LastName], [City], [Country] FROM [Employees]
UNION ALL
SELECT [ContactName], [City], [Country] FROM [Customers]
```

The plan for this query uses the concatenation operator:

```
|--Concatenation
   |--Compute Scalar(DEFINE:([Expr1003]=([Employees].[FirstName] + N' ')+
      [Employees].[LastName]))
   |     |--Clustered Index Scan(OBJECT:([Employees].[PK_Employees]))
   |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers]))
```

The concatenation operator simply executes each of its inputs—it may have more than two—one at a time and returns the results from each input. If we have any employees who also happen to be customers, these individuals will be returned twice by this query.

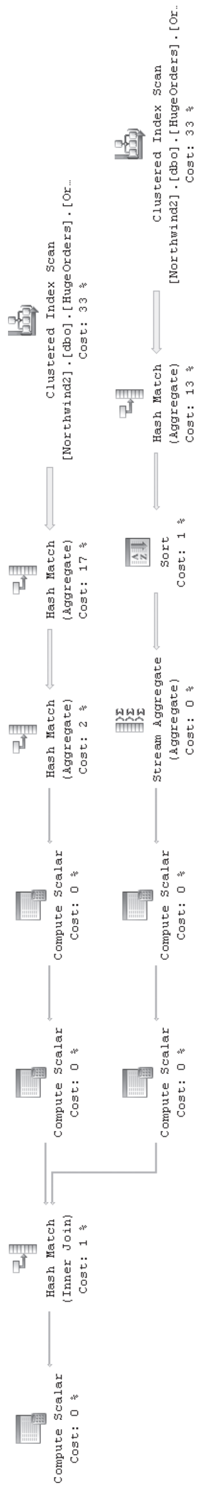


Figure 3-14 Graphical plan showing hash aggregation and a hash join to compute multiple distinct aggregates

Now let's consider the following similar query, which outputs a list of all cities and countries in which we have employees and/or customers:

```
SELECT [City], [Country] FROM [Employees]
UNION
SELECT [City], [Country] FROM [Customers]
```

Because this query uses UNION rather than UNION ALL, the query plan must eliminate duplicates. The optimizer uses the same concatenation operator as in the prior plan, but adds a sort distinct to eliminate duplicates:

```
|--Sort(DISTINCT ORDER BY:([Union1006] ASC, [Union1007] ASC))
   |--Concatenation
      |--Clustered Index Scan(OBJECT:([Employees].[PK_Employees]))
      |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers]))
```

Another essentially identical alternative plan is to replace the sort distinct with a hash aggregate. A sort distinct requires memory proportional to the number of input rows before the duplicates are eliminated, while a hash aggregate requires memory proportional to the number of output rows after the duplicates are eliminated. Thus, a hash aggregate requires less memory than the sort distinct when there are many duplicates and the optimizer is more likely to choose a hash aggregate when it expects many duplicates. For example, consider the following query, which combines data from the *Orders* and *BigOrders* tables to generate a list of all countries to which we have shipped orders:

```
SELECT [ShipCountry] FROM [Orders]
UNION
SELECT [ShipCountry] FROM [BigOrders]
```

Here is the query plan, which shows the hash distinct:

```
|--Hash Match(Aggregate, HASH:([Union1007]), RESIDUAL:(...))
   |--Concatenation
      |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders]))
      |--Clustered Index Scan(OBJECT:([BigOrders].[OrderID]))
```

Next suppose that we wish to find a list of all employees and customers sorted by name. We know if we can create appropriate indexes on the *Employees* and *Customers* tables that we can get a sorted list of employees or customers. Let's see what happens if we create both indexes and run a UNION ALL query with an ORDER BY clause. For this example, we'll need to create a new table with a subset of the columns from the *Employees* table and we'll need to create two new indexes. Here is the script to create the new table and indexes along with the test query:

```
SELECT [EmployeeId], [FirstName] + N' ' + [LastName] AS [ContactName],
   [City], [Country]
INTO [NewEmployees]
FROM [Employees]
```

```
ALTER TABLE [NewEmployees] ADD CONSTRAINT [PK_NewEmployees] PRIMARY KEY ([EmployeeId])
CREATE INDEX [ContactName] ON [NewEmployees]([ContactName])
CREATE INDEX [ContactName] ON [Customers]([ContactName])

SELECT [ContactName] FROM [NewEmployees]
UNION ALL
SELECT [ContactName] FROM [Customers]
ORDER BY [ContactName]
```

Here is the query plan:

```
|--Merge Join(Concatenation)
   |--Index Scan(OBJECT:([NewEmployees].[ContactName]), ORDERED FORWARD)
   |--Index Scan(OBJECT:([Customers].[ContactName]), ORDERED FORWARD)
```

Notice that, instead of a concatenation operator, we get a merge join (concatenation) operator. The merge join (concatenation) or merge concatenation operator is not really a join at all. It is implemented by the same iterator as the merge join, but it actually performs a UNION ALL (just like a regular concatenation operator) while preserving the order of the input rows. Like a merge join, a merge concatenation requires that input data be sorted on the merge key (in this case the *ContactName* column from the two input tables). By using the order-preserving merge concatenation operator instead of the non-order–preserving concatenation operator, the optimizer avoids the need to add an explicit sort to the plan.The graphical plan is shown in Figure 3-15.

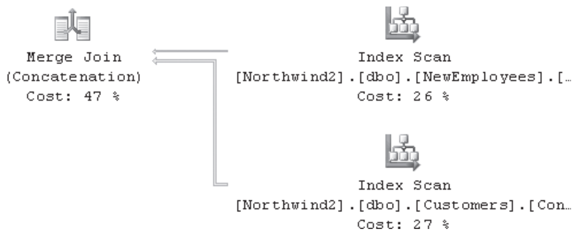


Figure 3-15 Graphical plan showing a merge concatenation

The merge join operator is capable of implementing both UNION ALL, as we have just seen, as well as UNION. For example, let's repeat the previous query as a UNION without the ORDER BY. In other words, we want to eliminate duplicates, but we are not interested in whether the results are sorted.

```
SELECT [ContactName] FROM [NewEmployees]
UNION
SELECT [ContactName] FROM [Customers]
```

The new plan still uses a merge join operator:

```
|--Merge Join(Union)
   |--Stream Aggregate(GROUP BY:([NewEmployees].[ContactName]))
   |    |--Index Scan(OBJECT:([NewEmployees].[ContactName]), ORDERED FORWARD)
   |--Stream Aggregate(GROUP BY:([Customers].[ContactName]))
        |--Index Scan(OBJECT:([Customers].[ContactName]), ORDERED FORWARD)
```

This time the plan has a merge join (union) or merge union operator. The merge union eliminates duplicate rows that appear in both of its inputs; it does *not* eliminate duplicate rows from either individual input. That is, if there is a name that appears in both the *NewEmployees* and the *Customers* tables, the merge union will eliminate that duplicate name. However, if there is a name that appears twice in the *NewEmployees* table or twice in the *Customers* table, the merge union by itself will *not* eliminate it. Thus, we see that the optimizer has also added stream aggregates above each index scan to eliminate any duplicates from the individual tables. As we discussed earlier, aggregation operators can be used to implement distinct operations. You can see the graphical plan in Figure 3-16.

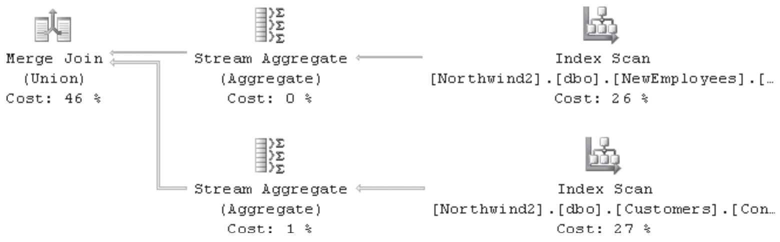


Figure 3-16 Graphical plan showing a merge union operation

Before continuing, let's drop the extra table and indexes that we created just for this example:

```
DROP TABLE [NewEmployees]
DROP INDEX [Customers].[ContactName]
```

There is one additional operator that SQL Server can use to perform a UNION. This operator is the hash union, and it is similar to a hash aggregate with two inputs. A hash union builds a hash table on its first input and eliminates duplicates from it just like a hash aggregate. It then reads its second input and, for each row, probes its hash table to see whether the row is a duplicate of a row from the first input. If the row is not a duplicate, the hash union returns it. Note that the hash union does not insert rows from the second input into the hash table. Thus, it does not eliminate duplicates that appear only in its second input. To use a hash union, the optimizer either must explicitly eliminate duplicates from the second input or must know that there are no duplicates in the second input.

Hash unions are rare. To see an example of a hash union, we need to create a large table with big rows but many duplicates. The following script creates two tables. The first table, *BigTable*, has 100,000 rows, and each row includes a *char(1000)* column, but all of the rows have the same value for the *Dups* column. The second table, *SmallTable*, has a uniqueness constraint to guarantee that there are no duplicates.

```
CREATE TABLE [BigTable] ([PK] int PRIMARY KEY, [Dups] int, [Pad] char(1000))
CREATE TABLE [SmallTable] ([PK] int PRIMARY KEY, [NoDups] int UNIQUE, [Pad] char(1000))

SET NOCOUNT ON
DECLARE @i int
SET @i = 0
BEGIN TRAN
WHILE @i < 100000
BEGIN
   INSERT [BigTable] VALUES (@i, 0, NULL)
   SET @i = @i + 1
   IF @i % 1000 = 0
   BEGIN
      COMMIT TRAN
      BEGIN TRAN
   END
END
COMMIT TRAN

SELECT [Dups], [Pad] FROM [BigTable]
UNION
SELECT [NoDups], [Pad] FROM [SmallTable]
```

The optimizer chooses a hash union for this query. The hash union is a good choice for eliminating the many duplicates from the *BigTable* table. Moreover, because of the uniqueness constraint on the *NoDups* column of the *SmallTable* table, there is no need to eliminate duplicates from this input to use it with a hash union.

Note On a machine with multiple processors or multiple cores, the optimizer may choose a substantially different parallel plan that does not include a hash union. If this happens, append an OPTION (MAXDOP 1) hint to the query and you should get the intended plan. The OPTION (MAXDOP 1) query hint forces SQL Server to choose a serial plan for the query. We will discuss parallelism later in this chapter and hints including this one in Chapters 4 and 5.

```
|--Hash Match(Union)
   |--Clustered Index Scan(OBJECT:([BigTable].[PK_BigTable]))
   |--Clustered Index Scan(OBJECT:([SmallTable].[PK_SmallTable]))
```

The graphical plan showing the hash union is shown in Figure 3-17.

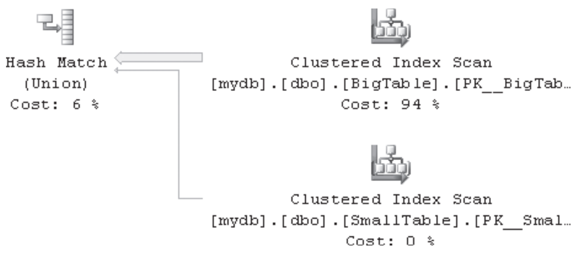


Figure 3-17 Graphical plan showing a hash union operation

Advanced Index Operations

Earlier in this chapter, we talked about index scans and seeks. You may have noticed that all of the index seek examples we've considered so far have involved simple predicates of the form *<column> <comparison operator> <expression>*. For instance, our very first example was *[OrderDate]* = '*1998-02-26*'. Let's take a look now at how SQL Server uses indexes to execute queries with AND'ed and OR'ed predicates. (AND'ed and OR'ed predicates are often referred to as "conjuctions" and "disjunctions," respectively.) Specifically, we'll look at dynamic index seeks, index unions, and index intersections.

Dynamic Index Seeks

Consider the following query with a simple IN list predicate:

```
SELECT [OrderId]
FROM [Orders]
WHERE [ShipPostalCode] IN (N'05022', N'99362')
```

We have an index on the *ShipPostalCode* column, and because this index covers the *OrderId* column, this query results in a simple index seek:

```
|--Index Seek(OBJECT:([Orders].[ShipPostalCode]), SEEK:([Orders].[ShipPostalCode]=N'05022'
   OR [Orders].[ShipPostalCode]=N'99362') ORDERED FORWARD)
```

Note that the IN list is logically identical to the OR'ed predicate in the index seek. SQL Server executes an index seek with OR'ed predicates by performing two separate index seek operations. First, the server executes an index seek with the *predicate [ShipPostalCode] = N*'*05022*' and then it executes a second index seek with the *predicate [ShipPostalCode] = N*'*99362*'.

Now consider the following identical query, which uses variables in place of constants:

```
DECLARE @SPC1 nvarchar(20), @SPC2 nvarchar(20)
SELECT @SPC1 = N'05022', @SPC2 = N'99362'
SELECT [OrderId]
FROM [Orders]
WHERE [ShipPostalCode] IN (@SPC1, @SPC2)
```

We might expect this query to result in a simple index seek as well. However, we instead get the following more complex plan:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1009],[Expr1010], [Expr1011]))
   |--Merge Interval
   |    |--Sort(TOP 2, ORDER BY:([Expr1012] DESC, [Expr1013] ASC,
           [Expr1009] ASC, [Expr1014] DESC))
   |       |--Compute Scalar (DEFINE:([Expr1012]=((4)&[Expr1011]) = (4) AND
              NULL = [Expr1009],[Expr1013]=(4)&[Expr1011],[Expr1014]=(16)&[Expr1011]))
   |          |--Concatenation
   |             |--Compute Scalar(DEFINE:([@SPC2]=[@SPC2], [@SPC2]=[@SPC2],
                    [Expr1003]=(62)))
   |                |--Constant Scan
   |             |--Compute Scalar(DEFINE:([@SPC1]=[@SPC1],
                    [@SPC1]=[@SPC1], [Expr1006]=(62)))
   |                |--Constant Scan
   |--Index Seek(OBJECT:([Orders].[ShipPostalCode]), SEEK:([Orders].[ShipPostalCode] >
[Expr1009] AND [Orders].[ShipPostalCode] < [Expr1010]) ORDERED FORWARD)
```

We do not get the index seek that we expect because the optimizer does not know the values of the variables at compile time and, therefore, cannot be sure whether they will have different values or the same value at runtime. If the variables have different values, the original simple index seek plan is valid. However, if the variables have the same value, the original plan is not valid. It would seek to the same index key twice and, thus, return each row twice. Obviously, this result would be incorrect as the query should only return each row once.

The more complex plan works by eliminating duplicates from the IN list at execution time. The two constant scans and the concatenation operator generate a "constant table" with the two IN list values. Then, the plan sorts the parameter values and the merge interval operator eliminates the duplicates (which, because of the sort, will be adjacent to one another). Finally, the nested loops join executes the index seek once for each unique value. You can see the graphical plan for this query in Figure 3-18.

You may be wondering why SQL Server needs the merge interval operator. Why couldn't the plan just use a sort distinct to eliminate any duplicate values? To answer this question, let's consider a slightly more complex query:

```
DECLARE @OD1 datetime, @OD2 datetime
SELECT @OD1 = '1998-01-01', @OD2 ='1998-01-04'
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN @OD1 AND DATEADD(day, 6, @OD1)
   OR [OrderDate] BETWEEN @OD2 AND DATEADD(day, 6, @OD2)
```

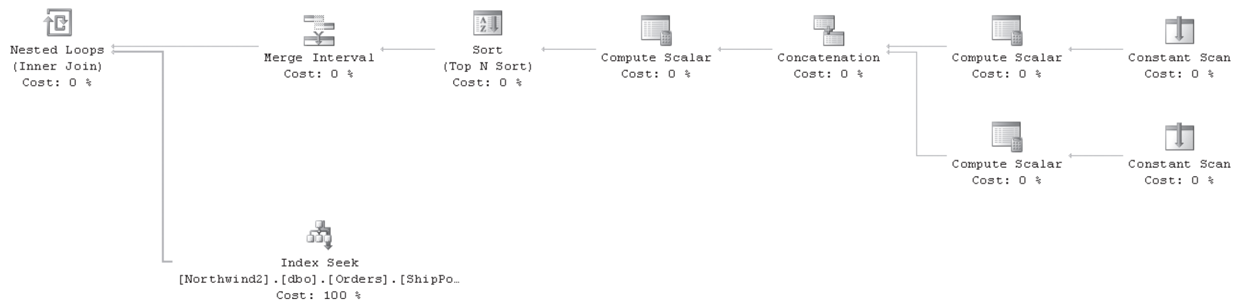


Figure 3-18 Graphical plan for a query with a dynamic seek for variables in an IN list

This query returns orders placed within one week of either of two dates. The query plan is nearly identical to the plan for the IN list query:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1011],[Expr1012],[Expr1013]))
   |--Merge Interval
   |    |--Sort(TOP 2, ORDER BY:([Expr1014] DESC, [Expr1015] ASC,
           [Expr1011] ASC, [Expr1016] DESC))
   |       |--Compute Scalar(DEFINE:([Expr1014]=((4)&[Expr1013]) = (4)
              AND NULL = [Expr1011], [Expr1015]=(4)&[Expr1013],
              [Expr1016]=(16)&[Expr1013]))
   |          |--Concatenation
   |             |--Compute Scalar(DEFINE:([@OD1]=[@OD1],
                    [ConstExpr1003]=dateadd(day,(6),[@OD1]),
                    [Expr1007]=(22)|(42)))
   |                  |--Constant Scan
   |             |--Compute Scalar(DEFINE:([@OD2]=[@OD2],
                    [ConstExpr1004]=dateadd(day,(6),[@OD2]),[Expr1010]=(22)|(42)))
   |                  |--Constant Scan
   |--Index Seek(OBJECT:([Orders].[OrderDate]),SEEK:([Orders].[OrderDate] > [Expr1011] AND
      [Orders].[OrderDate] < [Expr1012]) ORDERED FORWARD)
```

Once again, the plan includes a pair of constant scans and a concatenation operator. However, this time instead of returning discrete values from an IN list, the constant scans return ranges. Unlike the prior example, it is no longer sufficient simply to eliminate duplicates. Now the plan needs to handle ranges that are not duplicates but do overlap. The sort ensures that ranges that may overlap are adjacent to one another and the merge interval operator collapses overlapping ranges. In our example, the two date ranges (1998-01-01 to 1998-01-07 and 1998-01-04 to 1998-01-10) do in fact overlap, and the merge interval collapses them into a single range (1998-01-01 to 1998-01-10). The graphical plan for this query is shown in Figure 3-19.

We refer to these plans as dynamic index seeks since the range(s) that SQL Server actually fetches are not statically known at compile time and are determined dynamically during execution. Dynamic index seeks and the merge interval operator can be used for both OR'ed and AND'ed predicates, though they are most common for OR'ed predicates. AND'ed predicates can generally be handled by using one of the predicates, preferably the most selective predicate, as the seek predicate and then applying all remaining predicates as residuals. Recall that with OR'ed predicates we want the *union* of the set of rows that matches *any* of the predicates, whereas with AND'ed predicates we want the *intersection* of the set of rows that matches *all* of the predicates.

Index Unions

We've just seen how SQL Server can use an index seek even if we have an OR'ed predicate. Next, let's consider a slightly different query that OR's predicates on two different columns:

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN '1998-01-01' AND '1998-01-07'
   OR [ShippedDate] BETWEEN '1998-01-01' AND '1998-01-07'
```

Delaney_Ch03.fm Page 159 Thursday, August 9, 2007 5:26 PM

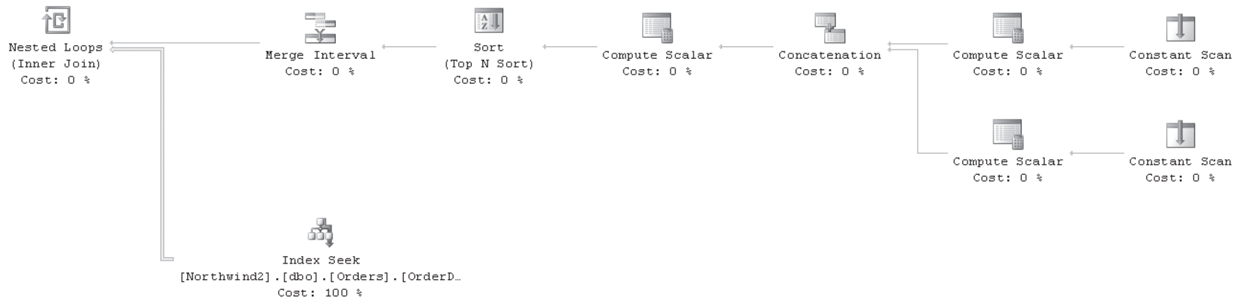


Figure 3-19 Graphical plan for a query with a dynamic seek for variables in a BETWEEN clause

This query returns orders that were either placed or shipped during the first week of 1998. We have indexes on both the *OrderDate* and the *ShippedDate* columns, but each of these indexes can only be used to satisfy one of the two predicates. If SQL Server uses the index on the *OrderDate* column to find orders placed during the first week of 1998, it may miss orders that were not placed but did ship during this week. Similarly, if it uses the index on the *ShippedDate* column to find orders shipped during the first week of 1998, it may miss orders that were placed but did not ship during this week.

It turns out that SQL Server has two strategies that it can use to execute a query such as this one. One option is to use a clustered index scan (or table scan) and apply the entire predicate to all rows in the table. This strategy is reasonable if the predicates are not very selective and if the query will end up returning many rows. However, if the predicates are reasonably selective and if the table is large, the clustered index scan strategy is not very efficient. The other option is to use both indexes. This plan looks as follows:

```
|--Sort(DISTINCT ORDER BY:([Orders].[OrderID] ASC))
   |--Concatenation
      |--Index Seek(OBJECT:([Orders].[OrderDate]),
         SEEK:([Orders].[OrderDate] >= '1998-01-01' AND
          [Orders].[OrderDate] <= '1998-01-07')
         ORDERED FORWARD)
      |--Index Seek(OBJECT:([Orders].[ShippedDate]),
         SEEK:([Orders].[ShippedDate] >= '1998-01-01' AND
         [Orders].[ShippedDate] <= '1998-01-07')
         ORDERED FORWARD)
```

The optimizer effectively rewrote the query as a union:

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] BETWEEN '1998-01-01' AND '1998-01-07'
UNION
SELECT [OrderId]
FROM [Orders]
WHERE [ShippedDate] BETWEEN '1998-01-01' AND '1998-01-07'
```

By using both indexes, this plan gets the benefit of the index seek—namely that it only retrieves rows that satisfy the query predicates—yet avoids missing any rows, as might happen if it used only one of the two indexes. However, by using both indexes, the plan may generate duplicates if, as is likely in this example, any of orders were both placed and shipped during the first week of 1998. To ensure that the query plan does not return any rows twice, the optimizer adds a sort distinct. We refer to this type of plan as an index union.

Note that the manual rewrite is only valid because the *OrderId* column is unique, which ensures that the UNION does not eliminate more rows than it should. However, even if we do not have a unique key and cannot manually rewrite the query, the optimizer always has an internal unique "relational key" for each row (the same key that it uses to perform bookmark lookups) and, thus, can always perform this transformation.

Next, let's consider the following nearly identical query:

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] = '1998-01-01'
   OR [ShippedDate] = '1998-01-01'
```

The only difference between this query and the prior one is that this query has equality predicates on both columns. Specifically, we are searching for orders that were either placed or shipped on the first day of 1998. Yet, this query yields the following plan:

```
|--Stream Aggregate(GROUP BY:([Orders].[OrderID]))
   |--Merge Join(Concatenation)
      |--Index Seek(OBJECT:([Orders].[OrderDate]),
         SEEK:([Orders].[OrderDate]='1998-01-01') ORDERED FORWARD)
      |--Index Seek(OBJECT:([Orders].[ShippedDate]),
         SEEK:([Orders].[ShippedDate]='1998-01-01') ORDERED FORWARD)
```

Instead of the concatenation and sort distinct operators, we now have a merge concatenation and a stream aggregate. Because we have equality predicates on the leading column of each index, the index seeks return rows sorted on the second column (the *OrderId* column) of each index. Since index seeks return sorted rows, this query is suitable for a merge concatenation. Moreover, since the merge concatenation returns rows sorted on the merge key (the *OrderId* column), the optimizer can use a stream aggregate instead of a sort to eliminate duplicates. This plan is generally a better choice because the sort distinct uses memory and could spill data to disk if it runs out of memory, while the merge concatenation and stream aggregate do not use memory.

Note that SQL Server did not use the merge concatenation in the prior example because the predicates were inequalities. The inequalities mean that the index seeks in that example returned rows sorted by the *OrderDate* and the *ShippedDate* columns. Because the rows were not sorted by the *OrderId* column, the optimizer could not use the merge concatenation without explicit sorts.

Although the above examples only involve two predicates and two indexes, SQL Server can use index union with any number of indexes, just as we can write a UNION query with any number of inputs.

A union only returns the columns that are common to all of its inputs. In each of the above index union examples (whether based on concatenation and sort distinct, merge union, or hash union), the only column that the indexes have in common is the clustering key *OrderId*. Thus, the union can only return the *OrderId* column. If we ask for other columns, the plan must perform a bookmark lookup. This is true even if one of the indexes in the union covers the extra columns. For example, consider the following query, which is identical to the above query but selects the *OrderDate* and *ShippedDate* columns in addition to the *OrderId* column:

```
SELECT [OrderId], [OrderDate], [ShippedDate]
FROM [BigOrders]
WHERE [OrderDate] = '1998-01-01'
   OR [ShippedDate] = '1998-01-01'
```

Note Note that we need to use the *BigOrders* table for this example as the optimizer tends to favor simply scanning the entire table instead of performing a bookmark lookup for smaller tables.

Here is the query plan showing the bookmark lookup:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Uniq1002],[BigOrders].[OrderID]))
   |--Stream Aggregate(GROUP BY:([BigOrders].[OrderID], [Uniq1002]))
   |    |--Merge Join(Concatenation)
   |        |--Index Seek(OBJECT:([BigOrders].[OrderDate]),
                 SEEK:([BigOrders].[OrderDate]='1998-01-01')
             ORDERED FORWARD)
   |        |--Index Seek(OBJECT:([BigOrders].[ShippedDate]),
                 SEEK:([BigOrders].[ShippedDate]='1998-01-01')
             ORDERED FORWARD)
   |--Clustered Index Seek(OBJECT:([BigOrders].[OrderID]),
      SEEK:([BigOrders].[OrderID]=[BigOrders].[OrderID] AND [Uniq1002]=[Uniq1002])
      LOOKUP ORDERED FORWARD)
```

Index Intersections

We've just seen how SQL Server can convert OR'ed predicates into a union query and use multiple indexes to execute this query. SQL Server can also use multiple indexes to execute queries with AND'ed predicates. For example, consider the following query:

```
SELECT [OrderId]
FROM [Orders]
WHERE [OrderDate] = '1998-02-26'
   AND [ShippedDate] = '1998-03-04'
```

This query is looking for orders that were placed on February 26, 1998, and were shipped on March 4, 1998. We have indexes on both the *OrderDate* and the *ShippedDate* columns, and we can use both indexes. Here is the query plan:

```
|--Merge Join(Inner Join, MERGE:([Orders].[OrderID])=([Orders].[OrderID]), RESIDUAL:(...))
   |--Index Seek(OBJECT:([Orders].[ShippedDate]),
      SEEK:([Orders].[ShippedDate]='1998-03-04')
      ORDERED FORWARD)
   |--Index Seek(OBJECT:([Orders].[OrderDate]),
      SEEK:([Orders].[OrderDate]='1998-02-26')
      ORDERED FORWARD)
```

This plan is very similar to the index union plan with the merge union operator, except that this time we have an actual join. Note that the merge join implements an inner join logical operation. It is really a join this time; it's not a union. The optimizer has effectively rewritten this query as a join (although the explicit rewrite does not get the same plan):

```
SELECT O1.[OrderId]
FROM [Orders] O1 JOIN [Orders] O2
```

```
   ON O1.[OrderId] = O2.[OrderId]
WHERE O1.[OrderDate] = '1998-02-26'
   AND O2.[ShippedDate] = '1998-03-04'
```

We refer to this query plan as an index intersection. Just as an index union can use different operators depending on the plan, so can index intersection. As in the merge union example, our first index intersection example uses a merge join because, as a result of the equality predicates, the two index seeks return rows sorted on the *OrderId* column. Now consider the following query which searches for orders that match a range of dates:

```
SELECT [OrderId]
FROM [BigOrders]
WHERE [OrderDate] BETWEEN '1998-02-01' AND '1998-02-04'
   AND [ShippedDate] BETWEEN '1998-02-09' AND '1998-02-12'
```

Note We again need to use the larger *BigOrders* table, as the optimizer favors a simple clustered index scan over a hash-join-based index intersection for the smaller table.

The inequality predicates mean the index seeks no longer return rows sorted by the *OrderId* column, therefore SQL Server cannot use a merge join without first sorting the rows. Instead of sorting, the optimizer chooses a hash join:

```
|--Hash Match(Inner Join, HASH:([BigOrders].[OrderID], [Uniq1002])=([BigOrders].[OrderID],
     [Uniq1002]), RESIDUAL:(...))
   |--Index Seek(OBJECT:([BigOrders].[OrderDate]),
      SEEK:([BigOrders].[OrderDate] >= '1998-02-01' AND
      [BigOrders].[OrderDate] <= '1998-02-04')
      ORDERED FORWARD)
   |--Index Seek(OBJECT:([BigOrders].[ShippedDate]),
      SEEK:([BigOrders].[ShippedDate] >= '1998-02-09' AND
      [BigOrders].[ShippedDate] <= '1998-02-12')
      ORDERED FORWARD)
```

Just like index union, SQL Server can use index intersection plans with more than two tables. Each additional table simply adds one more join to the query plan.

Unlike an index union, which can only deliver those columns that all of the indexes have in common, an index intersection can deliver all of the columns covered by any of the indexes. There is no need for a bookmark lookup. For example, the following query, which selects the *OrderDate* and *ShippedDate* columns in addition to the *OrderId* column, uses the same plan as the similar example that selected just the *OrderId* column:

```
SELECT [OrderId], [OrderDate], [ShippedDate]
FROM [Orders]
WHERE [OrderDate] = '1998-02-26'
   AND [ShippedDate] = '1998-03-04'
```

Subqueries

Subqueries are powerful tools that enable us to write far more expressive and far more complex queries. There are many different types of subqueries and many different ways to use subqueries. A complete discussion of subqueries could fill an entire chapter, if not an entire book. In this section, we'll take an introductory look at subqueries.

Subqueries are essentially joins. However, as we'll see, some subqueries generate more complex joins or use some fairly unusual join features.

Before we discuss specific examples, let's look at the different ways that we can classify subqueries. Subqueries can be categorized in three ways:

- Noncorrelated vs. correlated subqueries. A noncorrelated subquery has no dependencies on the outer query, can be evaluated independently of the outer query, and returns the same result for each row of the outer query. A correlated subquery does have a dependency on the outer query. It can only be evaluated in the context of a row from the outer query and may return a different result for each row of the outer query.
- Scalar vs. multirow subqueries. A scalar subquery returns or is expected to return a single row (that is, a scalar), whereas a multirow subquery may return a set of rows.
- The clause of the outer query in which the subquery appears. Subqueries can be used in nearly any context, including the SELECT list and the FROM, WHERE, ON, and HAVING clauses of the main query.

Noncorrelated Scalar Subqueries

Let's begin our discussion of subqueries by looking at some simple noncorrelated scalar subqueries. The following query returns a list of orders where the freight charge exceeds the average freight charge for all orders:

```
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] >
    (
    SELECT AVG(O2.[Freight])
    FROM [Orders] O2
    )
```

Notice that we could extract the calculation of the average freight charge and execute it as a completely independent query. Thus, this subquery is noncorrelated. Also, notice that this subquery uses a scalar aggregate and, thus, returns exactly one row. Thus, this subquery is also a scalar subquery. Let's examine the query plan:

```
|--Nested Loops(Inner Join, WHERE:([O1].[Freight]>[Expr1004]))
    |--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1011]=(0)
        THEN NULL
        ELSE
        [Expr1012]/CONVERT_IMPLICIT(money,[Expr1011],0) END))
   |    |--Stream Aggregate(DEFINE:([Expr1011]=COUNT_BIG([O2].[Freight]),
              [Expr1012]=SUM([O2].[Freight])))
   |       |--Clustered Index Scan
                 (OBJECT:([Orders].[PK_Orders] AS [O2]))
   |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O1]))
```

As you might expect, SQL Server executes this query by first calculating the average freight on the outer side of the nested loops join. The calculation requires a scan of the *Orders* table (alias [O2]). Since this calculation yields precisely one row, SQL Server then executes the scan on the *Orders* table (alias [O1]) on the inner side of the join exactly once. The average freight result calculated by the subquery (and stored in [Expr1004]) is used to filter the rows from the second scan.

Now, let's look at another noncorrelated scalar subquery. This time we want to find those orders placed by a specific customer that we've selected by name:

```
SELECT O.[OrderId]
FROM [Orders] O
WHERE O.[CustomerId] =
   (
   SELECT C.[CustomerId]
   FROM [Customers] C
   WHERE C.[ContactName] = N'Maria Anders'
   )
```

Notice that this time the subquery does not have a scalar aggregate to guarantee that it returns exactly one row. Moreover, there is no unique index on the *ContactName* column, so it is certainly possible that this subquery could actually return multiple rows. However, because the subquery is used in the context of an equality predicate, it is a scalar subquery and must return a single row. If we have two customers with the name "Maria Anders" (which we do not), this query must fail. SQL Server ensures that the subquery returns at most one row by counting the rows with a stream aggregate and then adding an assert operator to the plan:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Expr1006]))
   |--Assert(WHERE:(CASE WHEN [Expr1005]>(1) THEN (0) ELSE NULL END))
   |    |--Stream Aggregate(DEFINE:([Expr1005]=Count(*),
               [Expr1006]=ANY([C].[CustomerID])))
   |       |--Clustered Index Scan (OBJECT:([Customers].[PK_Customers] AS [C]),
                 WHERE:([C].[ContactName]=N'Maria Anders'))
   |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
      SEEK:([O].[CustomerID]=[Expr1006])
      ORDERED FORWARD)
```

If the assert operator finds that the subquery returned more than one row [that is, if *[Expr1005]>(1)* is true], it raises the following error:

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the
subquery follows =, !=, <, <= , >, >= or when the subquery
is used as an expression.
```

Note that SQL Server uses the assert operator to check many other conditions such as constraints (check, referential integrity, etc.), the maximum recursion level for common table expressions (CTEs), warnings for duplicate key insertions to indexes built with the IGNORE_DUP_KEY option, and more.

The ANY aggregate is a special internal-only aggregate that, as its name suggests, returns any row. Since this plan raises an error if the scan of the *Customers* table returns more than one row, the ANY aggregate has no real effect. The plan could as easily use the MIN or MAX aggregates and get the same result. However, some aggregate is necessary since the stream aggregate expects each output column either to be aggregated or in the GROUP BY clause (which is empty in this case). This is the same reason that the following query does not compile:

```
SELECT COUNT(*), C.[CustomerId]
FROM [Customers] C
WHERE C.[ContactName] = N'Maria Anders'
```

If you try to execute this query, you get the following error:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Customers.CustomerID' is invalid in the select list because
it is not contained in either an aggregate function or the
GROUP BY clause.
```

The assert operator is not expensive per se and is relatively harmless in this simple example, but it does limit the set of transformations available to the optimizer, which may result in an inferior plan in some cases. Often, creating a unique index or rewriting the query eliminates the assert operator and improves the query plan. For example, as long as we are not concerned that there might be two customers with the same name, the above query can be written as a simple join:

```
SELECT O.[OrderId]
FROM [Orders] O JOIN [Customers] C
   ON O.[CustomerId] = C.[CustomerId]
WHERE C.[ContactName] = N'Maria Anders'
```

This query produces a much simpler join plan:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
   |--Clustered Index Scan(OBJECT:([Customers].[PK_Customers] AS [C]),
      WHERE:([C].[ContactName]=N'Maria Anders'))
   |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
      SEEK:([O].[CustomerID]=[C].[CustomerID]) ORDERED FORWARD)
```

While writing this query as a simple join is the best option, let's see what happens if we create a unique index on the *ContactName* column:

```
CREATE UNIQUE INDEX [ContactName] ON [Customers] ([ContactName])

SELECT O.[OrderId]
FROM [Orders] O
```

```
WHERE O.[CustomerId] =
   (
   SELECT C.[CustomerId]
   FROM [Customers] C
   WHERE C.[ContactName] = N'Maria Anders'
   )
DROP INDEX [Customers].[ContactName]
```

Because of the unique index, the optimizer knows that the subquery can produce only one row, eliminates the now unnecessary stream aggregate and assert operators, and converts the query into a join:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
   |--Index Seek(OBJECT:([Customers].[ContactName] AS [C]),
      SEEK:([C].[ContactName]=N'Maria Anders') ORDERED FORWARD)
   |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
      SEEK:([O].[CustomerID]=[C].[CustomerID]) ORDERED FORWARD)
```

Correlated Scalar Subqueries

Now that we've seen how SQL Server evaluates a simple noncorrelated subquery, let's explore what happens if we have a correlated scalar subquery. The following query is similar to the first subquery we tried, but this time it returns those orders in which the freight charge exceeds the average freight charge for all previously placed orders:

```
SELECT O1.[OrderId]
FROM [Orders] O1
WHERE O1.[Freight] >
   (
   SELECT AVG(O2.[Freight])
   FROM [Orders] O2
   WHERE O2.[OrderDate] < O1.[OrderDate]
   )
```

This time SQL Server cannot execute the subquery independently. Because of the correlation on the *OrderDate* column, the subquery returns a different result for each row from the main query. Recall that with the noncorrelated subquery, SQL Server evaluated the subquery first and then executed the main query. This time SQL Server evaluates the main query first and then evaluates the subquery once for each row from the main query:

```
|--Filter(WHERE:([O1].[Freight]>[Expr1004]))
   |--Nested Loops(Inner Join, OUTER REFERENCES:([O1].[OrderDate]))
      |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O1]))
         |--Index Spool(SEEK:([O1].[OrderDate]=[O1].[OrderDate]))
         |--Compute Scalar(DEFINE:([Expr1004]=
            CASE WHEN [Expr1011]=(0)
            THEN NULL
            ELSE [Expr1012]
            /CONVERT_IMPLICIT(money,[Expr1011],0) END))
            |--Stream Aggregate (DEFINE:([Expr1011]=
               COUNT_BIG([O2].[Freight]),[Expr1012]=SUM([O2].[Freight])))
```

```
|--Index Spool(SEEK:([O2].[OrderDate] <[O1].[OrderDate]))
|--Clustered Index Scan
(OBJECT:([Orders].[PK_Orders] AS [O2]))
```

This plan is not as complicated as it looks. You can see the graphical plan in Figure 3-20. The index spool immediately above the scan of [O2] is an eager index spool or index-on-the-fly spool. It builds a temporary index on the *OrderDate* column of the *Orders* table. It is called an eager spool because it "eagerly" loads its entire input set and builds the temporary index as soon as it is opened.

The index makes subsequent evaluations of the subquery more efficient since there is a predicate on the *OrderDate* column. The stream aggregate computes the average freight charge for each execution of the subquery. The index spool above the stream aggregate is a lazy index spool. It merely caches subquery results. If it encounters any *OrderDate* a second time, it returns the cached result rather than recomputing the subquery. It is called a lazy spool because it "lazily" loads results on demand only. Finally, the filter at the top of the plan compares the freight charge for each order to the subquery result ([Expr1004]) and returns those rows that qualify.

Note We can determine the types of the spools more easily from the complete SHOWPLAN_ALL output or from the graphical plan. The logical operator for the spool indicates whether each spool is an eager or lazy spool. The spools are there strictly for performance. The optimizer decides whether to include them in the plan based on its cost estimates. If the optimizer does not expect many duplicate values for the *OrderDate* column or many rows at all from the outer side of the join, it may eliminate the spools. For example, if you try the same query with a selective filter on the main query such as *O1.[ShipCity] = N'Berlin'*, the spools go away.

We have already seen how a nested loops join executes its inner input once for each row from its outer input. In most cases, each execution of the inner input proceeds completely independently of any prior executions. However, spools are special. A spool, such as the lazy index spool in the above example, is designed to optimize the case in which the inner side of the join executes with the same correlated parameter(s) multiple times. Thus, for a spool, it is useful to distinguish between executions with the same correlated parameter(s) or "rewinds" and executions with different correlated parameters or "rebinds." Specifically, a rewind is defined as an execution with the same correlated parameter(s) as the immediately preceding execution, whereas a rebind is defined as an execution with different correlated parameters than the immediately preceding execution.

A rewind results in the spool playing back a cached result, while a rebind results in the spool "binding" new correlated parameter values and loading a new result. A regular (nonindex) lazy spool only caches one result set at a time. Thus, a regular spool truncates and "reloads" its worktable on each rebind. A lazy index spool, such as the spool in the above plan, accumulates results and does not truncate its worktable on a rebind.

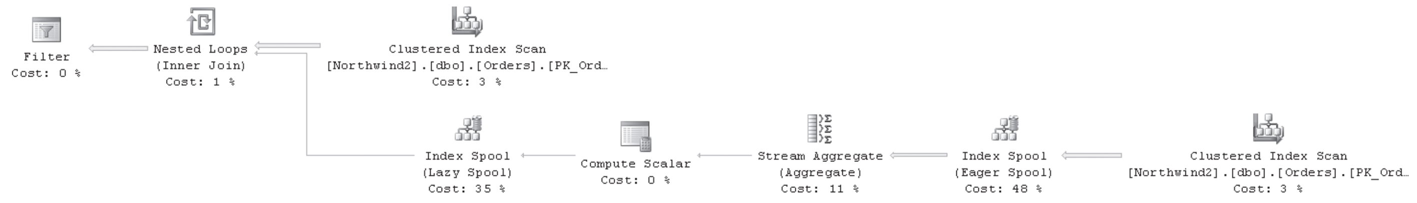
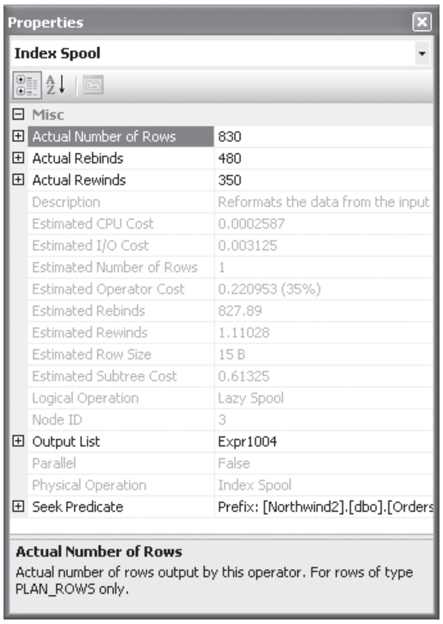


Figure 3-20   Graphical plan showing two index spool operations: one lazy and one eager

We can see the estimated and actual number of rewinds and rebinds using SET STATISTICS XML ON or using the graphical plan. For example, here are the statistics XML for the lazy index spool. Notice that, as you might expect, the sum of the number of rewinds and rebinds is the same as the total number of executions

```
<RelOp NodeId="3" PhysicalOp="Index Spool" LogicalOp="Lazy Spool"...
       EstimateRebinds="827.89" EstimateRewinds="1.11028">
   <RunTimeInformation>
   <RunTimeCountersPerThread Thread="0" ActualRows="830" ActualRebinds="480"
          ActualRewinds="350"
          ActualEndOfScans="0"
          ActualExecutions="830" />
   </RunTimeInformation>
</RelOp>
```

The ToolTip from the graphical plan, showing the same information as in the above XML fragment, is shown in Figure 3-21.


Properties

Index Spool

⊟ Misc

⊞ Actual Number of Rows	830
⊞ Actual Rebinds	480
⊞ Actual Rewinds	350
Description	Reformats the data from the input
Estimated CPU Cost	0.0002587
Estimated I/O Cost	0.003125
Estimated Number of Rows	1
Estimated Operator Cost	0.220953 (35%)
Estimated Rebinds	827.89
Estimated Rewinds	1.11028
Estimated Row Size	15 B
Estimated Subtree Cost	0.61325
Logical Operation	Lazy Spool
Node ID	3
⊞ Output List	Expr1004
Parallel	False
Physical Operation	Index Spool
⊞ Seek Predicate	Prefix: [Northwind2].[dbo].[Orders

Actual Number of Rows
Actual number of rows output by this operator. For rows of type PLAN_ROWS only.

Figure 3-21 ToolTip showing rewind and rebind information

Note that rewinds and rebinds are counted the same way for index and nonindex spools. As described previously, a reexecution is counted as a rewind only if the correlated parameter(s) remain the same as the immediately prior execution and is counted as a rebind if the correlated parameter(s) change from the prior execution. This is true even for reexecutions, in which the same correlated parameter(s) were encountered in an earlier, though not the immediately prior, execution. However, since lazy index spools like the one in this example retain results for all prior executions and all previously encountered correlated parameter values, the spool may treat some reported rebinds as rewinds. In other words, by failing to account for correlated parameter(s) that were seen prior to the most recent execution, the query plan statistics may overreport the number of rebinds for an index spool.

Next let's look at another example of a correlated scalar subquery. Suppose that we wish to find those orders for which the freight charge exceeds the average freight charge for all orders placed by the same customer:

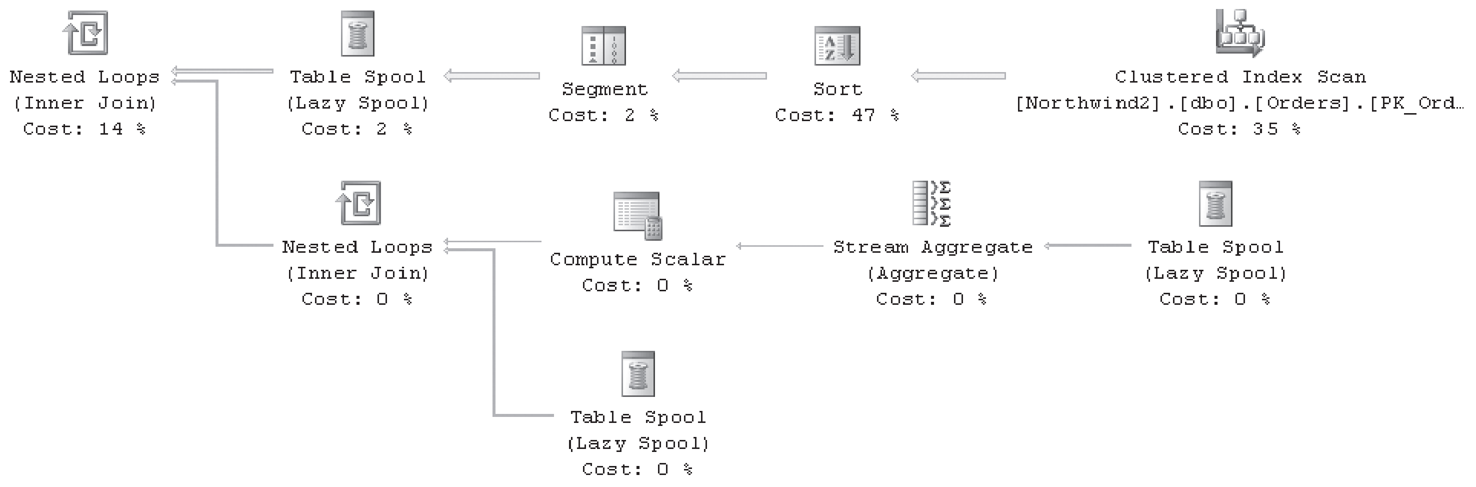
```
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] >
   (
   SELECT AVG(O2.[Freight])
   FROM [Orders] O2
   WHERE O2.[CustomerId] = O1.[CustomerId]
   )
```

This query is very similar to the previous one, yet we get a substantially different plan:

```
|--Nested Loops(Inner Join)
   |--Table Spool
   |    |--Segment
   |       |--Sort(ORDER BY:([O1].[CustomerID] ASC))
   |          |--Clustered Index Scan (OBJECT:([Orders].[PK_Orders] AS [O1]),
                 WHERE:([O1].[CustomerID] IS NOT NULL))
   |--Nested Loops(Inner Join, WHERE:([O1].[Freight]>[Expr1004]))
        |--Compute Scalar(DEFINE:([Expr1004]=
                CASE WHEN [Expr1012]=(0)
                  THEN NULL
                  ELSE [Expr1013]
                  /CONVERT_IMPLICIT(money,[Expr1012],0)
                END))
   |    |--Stream Aggregate(DEFINE:([Expr1012]=
          COUNT_BIG([O1].[Freight]),[Expr1013]=SUM([O1].[Freight])))
   |    |  |--Table Spool
   |    |--Table Spool
```

Again, this plan is not as complicated as it looks and you can see the graphical plan in Figure 3-22. The outer side of the topmost nested loops join sorts the rows of the clustered index scan by the *CustomerId* column. The segment operator breaks the rows into groups (or segments) with the same value for the *CustomerId* column. Since the rows are sorted, sets of rows with the same *CustomerId* value will be consecutive. Next, the table spool—a segment spool—reads and saves one of these groups of rows that share the same *CustomerId* value.

When the spool finishes loading a group of rows, it returns a single row for the entire group. (Note that a segment spool is the only type of spool that exhibits this behavior of returning only a single row regardless of how many input rows it reads.) At this point, the topmost nested loops join executes its inner input. The two leaf-level table spools—secondary spools—replay the group of rows that the original segment spool saved. The graphical plan illustrates the relationship between the secondary and primary spools by showing that the Primary Node ID for each of the secondary spools is the same as the Node ID for the primary segment



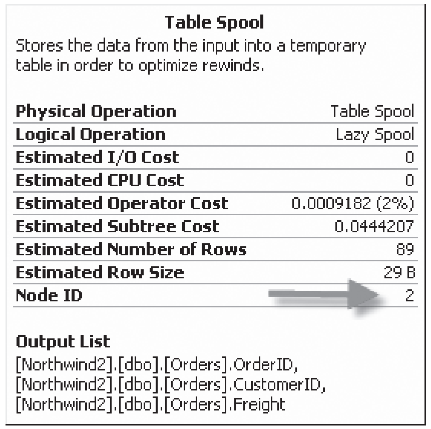


Table Spool

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

Physical Operation	Table Spool
Logical Operation	Lazy Spool
Estimated I/O Cost	0
Estimated CPU Cost	0
Estimated Operator Cost	0.0009182 (2%)
Estimated Subtree Cost	0.0444207
Estimated Number of Rows	89
Estimated Row Size	29 B
Node ID	2

Output List

[Northwind2].[dbo].[Orders].OrderID, [Northwind2].[dbo].[Orders].CustomerID, [Northwind2].[dbo].[Orders].Freight

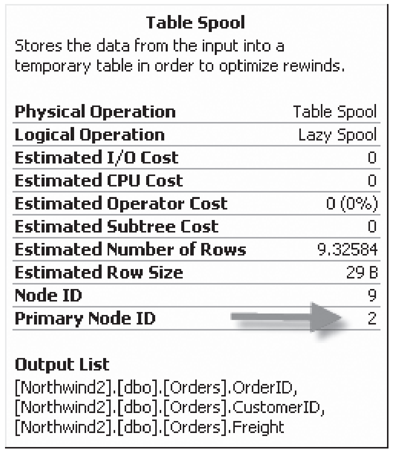


Table Spool

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

Physical Operation	Table Spool
Logical Operation	Lazy Spool
Estimated I/O Cost	0
Estimated CPU Cost	0
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0
Estimated Number of Rows	9.32584
Estimated Row Size	29 B
Node ID	9
Primary Node ID	2

Output List

[Northwind2].[dbo].[Orders].OrderID, [Northwind2].[dbo].[Orders].CustomerID, [Northwind2].[dbo].[Orders].Freight

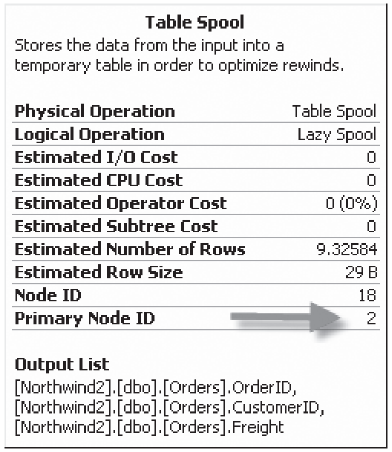


Table Spool

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

Physical Operation	Table Spool
Logical Operation	Lazy Spool
Estimated I/O Cost	0
Estimated CPU Cost	0
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0
Estimated Number of Rows	9.32584
Estimated Row Size	29 B
Node ID	18
Primary Node ID	2

Output List

[Northwind2].[dbo].[Orders].OrderID, [Northwind2].[dbo].[Orders].CustomerID, [Northwind2].[dbo].[Orders].Freight

Figure 3-22 Graphical plan showing a segment spool with two secondary spools

spool. (In this example, the primary spool's Node ID is 2.) The stream aggregate computes the average freight for each group of rows (and, thus, for each *CustomerId* value). The result of the stream aggregate is a single row. The inner of the two nested loops compares each spooled row (which again consists of rows with the same *CustomerId* value) against this average and returns those rows that qualify. Finally, the segment spool truncates its worktable and repeats the process beginning with reading the next group of rows with the next *CustomerId* value.

By using the segment spool, the optimizer creates a plan that needs to scan the *Orders* table only one time. We refer to a spool such as this one that replays the same set of rows in different places within the plan as a common subexpression spool. Note that not all common subexpression spools are segment spools.

Note We know that the spool is a segment spool, as it appears immediately above a segment operator in the plan. As in the prior example, we can find more information about the spools and other operators in this plan from SHOWPLAN_ALL, SHOWPLAN_XML, or the graphical plan. SHOWPLAN_XML and the graphical plan provide the richest information including the *group by* column for the segment operator and the primary spool's Node ID for each secondary spool.

Finally, suppose that we want to compute the order with the maximum freight charge placed by each customer:

```
SELECT O1.[OrderId], O1.[Freight]
FROM [Orders] O1
WHERE O1.[Freight] =
    (
    SELECT MAX(O2.[Freight])
    FROM [Orders] O2
    WHERE O2.[CustomerId] = O1.[CustomerId]
    )
```

This query is similar to the previous two queries, yet we once again get a very different and surprisingly simple plan:

```
|--Top(TOP EXPRESSION:((1)))
    |--Segment
        |--Sort(ORDER BY:([O1].[CustomerID] DESC, [O1].[Freight] DESC))
            |--Clustered Index Scan
                (OBJECT:([Orders].[PK_Orders] AS [O1]),
                 WHERE:([O1].[Freight] IS NOT NULL AND
                 [O1].[CustomerID] IS NOT NULL))
```

This plan sorts the *Orders* table by the *CustomerId* and *Freight* columns. As in the previous example, the segment operator breaks the rows into groups or segments with the same value for the *CustomerId* column. The top operator is a segment top. Unlike a normal top, which returns the top *N* rows for the entire input set, a segment top returns the top *N* rows for each group. The top is also a "top with ties." A top with ties returns more than *N* rows if there are duplicates or ties for the *N*th row. In this query plan, since the sort assures that the rows with the maximum freight charge are ordered first within each group, the top returns the row or rows with the maximum freight charge from each group. This plan is very efficient since it processes the *Orders* table only once and, unlike the previous plan, does not need a spool.

Note As with the segment spool, we know that the top is a segment top because it appears immediately above a segment operator. We can also tell that the top is a top with ties by checking SHOWPLAN_ALL, SHOWPLAN_XML, or the graphical plan.

Removing Correlations

The SQL Server query optimizer is able to remove correlations from many subqueries including scalar and multirow queries. If the optimizer is unable to remove correlations from a subquery, it must execute the subquery plan on the inner side of a nested loops join. However, by removing correlations, the optimizer can transform a subquery into a regular join and consider more plan alternatives with different join orders and join types. For example, the following query uses a noncorrelated subquery to return orders placed by customers who live in London:

```
SELECT O.[OrderId]
FROM [Orders] O
WHERE O.[CustomerId] IN
    (
    SELECT C.[CustomerId]
    FROM [Customers] C
    WHERE C.[City] = N'London'
    )
```

We can easily write the same query using a correlated subquery:

```
SELECT O.[OrderId]
FROM [Orders] O
WHERE EXISTS
    (
    SELECT *
    FROM [Customers] C
    WHERE C.[CustomerId] = O.[CustomerId]
        AND C.[City] = N'London'
    )
```

One of these queries includes a noncorrelated subquery whereas the other includes a correlated subquery, however the optimizer generates the same plan for both queries:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([C].[CustomerID]))
    |--Index Seek(OBJECT:([Customers].[City] AS [C]),
        SEEK:([C].[City]=N'London')
        ORDERED FORWARD)
    |--Index Seek(OBJECT:([Orders].[CustomerID] AS [O]),
        SEEK:([O].[CustomerID]= [C].[CustomerID])
        ORDERED FORWARD)
```

In the case of the second query, the optimizer removes the correlation from the subquery so that it can scan the *Customers* table first (on the outer side of the nested loops join). If the optimizer did not remove the correlation, the plan would need to scan the *Orders* table first to generate a *CustomerId* value before it could scan the *Customers* table.

For a more complex example of subquery decorrelation, consider the following query, which outputs a list of orders along with the freight charge for each order and the average freight charge for all orders by the same customer:

```
SELECT O1.[OrderId], O1.[Freight],
(
   SELECT AVG(O2.[Freight])
   FROM [Orders] O2
   WHERE O2.[CustomerId] = O1.[CustomerId]
   ) Avg_Freight
FROM [Orders] O1
```

We might expect that a correlated SELECT list subquery, like the one used in this query, must be evaluated exactly once for each row from the main query. However, as the following plan illustrates, the optimizer is able to remove the correlation from this query:

```
|--Compute Scalar(DEFINE:([Expr1004]=[Expr1004]))
   |--Hash Match(Right Outer Join,HASH:([O2].[CustomerID])=([O1].[CustomerID]),
      RESIDUAL:(...))
      |--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1013]=(0)
         THEN NULL
         ELSE [Expr1014]
         /CONVERT_IMPLICIT(money,[Expr1013],0)
         END))
      |    |--Stream Aggregate(GROUP BY:([O2].[CustomerID])
              DEFINE:([Expr1013]=COUNT_BIG([O2].[Freight]),
                    [Expr1014]=SUM([O2].[Freight])))
      |       |--Sort(ORDER BY:([O2].[CustomerID] ASC))
      |          |--Clustered Index Scan
                    (OBJECT:([Orders].[PK_Orders] AS [O2]))
      |--Clustered Index Scan(OBJECT:([Orders].[PK_Orders] AS [O1]))
```

This plan first computes the average freight charge for all customers and then joins this result with the *Orders* table. Note how this plan computes the average freight charge for each customer exactly once regardless of the number of orders placed by that customer. Had the optimizer not removed the correlation, the plan would have had to compute the average freight charge for each customer separately for each order placed by that customer. For example, if a customer placed three orders, the plan would have computed the average freight charge for that customer three times. Clearly, the plan with the decorrelated subquery is more efficient. On the other hand, if we have a sufficiently selective predicate on the main query, the correlated plan does become more efficient and the optimizer will select it.

In addition to computing the average freight charges only once per customer, by removing the correlation, the optimizer is free to use any join operator. For example, this plan uses a hash join. The join itself is a right outer join. The outer join guarantees that the plan returns all orders, even those that might have a NULL value for the *CustomerId* column. An inner join would discard such rows since NULLs never join.

Subqueries in CASE Expressions

Normally, SQL Server evaluates CASE expressions like any other scalar expression, often using a compute scalar operator. In the absence of any subqueries, there is really nothing remarkable about a CASE expression. However, it is possible to use subqueries in the WHEN, THEN, and ELSE clauses of a CASE expression. SQL Server uses some slightly more exotic join functionality, which we have not seen yet, to evaluate CASE expressions with subqueries. To see how these plans work, we'll use the following script to set up an artificial scenario:

```
CREATE TABLE [MainTable] ([PK] int PRIMARY KEY, [Col1] int, [Col2] int, [Col3] int)
CREATE TABLE [WhenTable] ([PK] int PRIMARY KEY, [Data] int)
CREATE TABLE [ThenTable] ([PK] int PRIMARY KEY, [Data] int)
CREATE TABLE [ElseTable] ([PK] int PRIMARY KEY, [Data] int)

INSERT [MainTable] VALUES (1, 11, 101, 1001)
INSERT [MainTable] VALUES (2, 12, 102, 1002)
INSERT [WhenTable] VALUES (11, NULL)
INSERT [ThenTable] VALUES (101, 901)
INSERT [ElseTable] VALUES (102, 902)
SELECT M.[PK],
   CASE WHEN EXISTS (SELECT * FROM [WhenTable] W WHERE W.[PK] = M.[Col1])
      THEN (SELECT T.[Data] FROM [ThenTable] T WHERE T.[PK] = M.[Col2])
      ELSE (SELECT E.[Data] FROM [ElseTable] E WHERE E.[PK] = M.[Col3])
   END AS Case_Expr
FROM [MainTable] M

DROP TABLE [MainTable], [WhenTable], [ThenTable], [ElseTable]
```

Semantically, this query scans table *MainTable* and for each row checks whether there is a matching row in table *WhenTable*. If there is a match, it looks up an output value in table *ThenTable*; otherwise, it looks up an output value in table *ElseTable*. To show precisely what is happening, the script also adds a few rows of data. *MainTable* has two rows. One of these rows matches a row in *WhenTable*, whereas the other does not. Thus, one row results in a lookup from *ThenTable* while the other row results in a lookup from *ElseTable*. Finally, *ThenTable* includes rows that match both *MainTable* rows, while *ElseTable* is empty. Although *ElseTable* is empty, the query still outputs all rows from *MainTable,* including the row that does not have a match in *WhenTable*. This row simply outputs NULL for the result of the CASE expression. Here is the output of the query:

```
PK                Case_Expr
-----------       -----------
1                 901
2                 NULL
```

And here is the query plan:

```
Rows Executes
0    0    |--Compute Scalar(DEFINE:([Expr1011]=CASE WHEN [Expr1012]
               THEN [T].[Data] ELSE [E].[Data] END))
2   1        |--Nested Loops(Left Outer Join, PASSTHRU:([Expr1012]),
                OUTER REFERENCES:([M].[Col3]))
2   1          |--Nested Loops(Left Outer Join,PASSTHRU:(IsFalseOrNull [Expr1012]),
                   OUTER REFERENCES:([M].[Col2]))
2   1          |    |--Nested Loops(Left Semi Join,
                         OUTER REFERENCES:([M].[Col1]),DEFINE:([Expr1012] = [PROBE
                              VALUE]))
2   1          |    |    |--Clustered Index Scan (OBJECT:([MainTable].[PK_MainTable]
                              AS [M]))
1   2          |    |    |--Clustered Index Seek(OBJECT:([WhenTable].[PK_WhenTable]
                              AS [W]),SEEK:([W].[PK]=[M].[Col1]))
1   1          |    |--Clustered Index Seek (OBJECT:([ThenTable].[PK_ThenTable] AS [T]),
                        SEEK:([T].[PK]=[M].[Col2]))
0   1          |--Clustered Index Seek (OBJECT:([ElseTable].[PK_ElseTable] AS [E]),
                   SEEK:([E].[PK]=[M].[Col3]))
```

As we might expect, this query plan begins by scanning *MainTable*. This scan returns two rows. Next, the plan executes the WHEN clause of the CASE expression. The plan implements the EXISTS subquery using a left semi-join with *WhenTable*. SQL Server frequently uses semi-joins to evaluate EXISTS subqueries since the semi-join merely checks whether a row from one input joins with or matches any row from the other input. However, a normal semi-join (or anti-semi-join) only returns rows for matches (or nonmatches). In this case, the query must return all rows from *MainTable,* regardless of whether these rows have a matching row in *WhenTable*. Thus, the semi-join cannot simply discard a row from *MainTable* just because *WhenTable* has no matching row.

The solution is a special type of semi-join with a PROBE column. This semi-join returns all rows from *MainTable* whether or not they match and sets the PROBE column (in this case [Expr1012]) to true or false to indicate whether or not it found a matching row in *WhenTable*. Since the semi-join does not actually return a row from *WhenTable*, without the PROBE column there would be no way to determine whether or not the semi-join found a match.

Next, depending on the value of the PROBE column, the query plan needs to look for a matching row in either *ThenTable* or *ElseTable*. However, the query plan must look in only one of the two tables. It cannot look in both. The plan uses a special type of nested loops join to ensure that it performs only one of the two lookups. This nested loops join has a special predicate known as a PASSTHRU predicate. The join evaluates the PASSTHRU predicate on each outer row. If the PASSTHRU predicate evaluates to true, the join immediately returns the outer row without executing its inner input. If the PASSTHRU predicate evaluates to false, the join proceeds normally and tries to join the outer row with an inner row.

Note In this example, the query plan could execute both subqueries (for the THEN and ELSE clauses of the CASE expression) and then discard any unnecessary results. However, besides being inefficient, in some cases executing the extra subqueries could cause the query plan to fail. For example, if one of the scalar subqueries joined on a nonunique column, it could return more than one row, which would result in an error. It would be incorrect for the plan to fail while executing an unnecessary operation.

The plan actually has two nested loops joins with PASSTHRU predicates: one to evaluate the THEN clause subquery and one to evaluate the ELSE clause subquery. The PASSTHRU predicate for the first (bottommost) join tests the condition *IsFalseOrNull [Expr1012]*. The IsFalseOrNull function simply inverts the Boolean PROBE column. For each *MainTable* row, if the semi-join finds a matching row, the PROBE column ([Expr1012]) is true, the PASSTHRU predicate evalutes to false, and the join evaluates the index seek on *ThenTable*. However, if the semi-join does not find a matching row, the PROBE column is false, the PASSTHRU predicate evalutes to true, and the join returns the *MainTable* row without evaluating the index seek on *ThenTable*. The PASSTHRU predicate for the next (topmost) join tests the opposite condition to determine whether to perform the index seek on *ElseTable*. Thus, the plan executes exactly one of the two index seeks for each *MainTable* row.

We can see the behavior of the PASSTHRU predicates by observing that while the *MainTable* scan and each of the joins returns two rows, the plan executes the index seeks on *ThenTable* and *ElseTable* only once. A PASSTHRU predicate is the only scenario in which the number of rows on the outer side of a nested loops join does not precisely match the number of executes on the inner side.

Also notice how the query plan uses outer joins since there is no guarantee that the THEN or ELSE subqueries will actually return any rows. In fact, in this example, the index seek on *ElseTable* is executed for one of the *MainTable* rows yet returns no rows. The outer join simply returns a NULL and the query still returns the *MainTable* row. If the query plan had used an inner join, it would have incorrectly discarded the *MainTable* row.

Tip The above query plan was generated using SQL Server 2005. If you run this example on SQL Server 2000, you will still get a plan with a PASSTHRU predicate, but it will appear in the plan as a regular WHERE clause predicate. Unfortunately, on SQL Server 2000, there is no easy way to differentiate a regular WHERE clause predicate from a PASSTHRU predicate.

This example demonstrates a CASE expression with a single WHEN clause. SQL Server supports CASE expressions with multiple WHEN clauses and multiple THEN clause subqueries in the same way. The PASSTHRU predicates merely get progressively more complex to ensure that only one of the THEN clauses (or the ELSE clause) is actually executed.

Parallelism

SQL Server has the ability to execute queries using multiple CPUs simultaneously. We refer to this capability as parallel query execution. Parallel query execution can be used to reduce the response time of (that is, speed up) a large query. It can also be used to a run a bigger query (one that processes more data) in about the same amount of time as a smaller query (that is, scale up) by increasing the number of CPUs used in processing the query.

While parallelism can be used to reduce the response time of a single query, this speedup comes at a cost: It increases the overhead associated with executing a query. While this overhead is relatively small, it does make parallelism inappropriate for small queries (e.g., for OLTP queries) in which the overhead would dominate the total execution time and the goal is to run the maximum number of concurrent queries and to maximize the overall throughput of the system. SQL Server does generally scale well; however, if we compare the same query running serially (that is, without parallelism and on a single CPU) and in parallel on two CPUs, we will typically find that the parallel execution time is more than half of the serial execution time. Again, this effect is caused by the parallelism overhead.

Parallelism is primarily useful on servers running a relatively small number of concurrent queries. On this type of server, parallelism can enable a small set of queries to keep many CPUs busy. On servers running many concurrent queries (such as an OLTP system), we do not need parallelism to keep the CPUs busy; the mere fact that we have so many queries to execute can keep the CPUs busy. As we've already discussed, running these queries in parallel would just add overhead that would reduce the overall throughput of the system.

SQL Server parallelizes queries by horizontally partitioning the input data into approximately equal-sized sets, assigning one set to each CPU, and then performing the same operation (for instance, aggregate, join, etc.) on each set. For example, suppose that SQL Server decides to use two CPUs to execute a hash aggregate that happens to be grouping on an integer column. The server creates two threads (one for each CPU). Each thread executes the same hash aggregate operator. SQL Server might partition the input data by sending rows in which the GROUP BY column is odd to one thread and rows in which the GROUP BY column is even to the other thread, as illustrated in Figure 3-23. As long as all rows that belong to one group are processed by one hash aggregate operator and one thread, the plan produces the correct result.

This method of parallel query execution is both simple and scales well. In the above example, both hash aggregate threads execute independently. The two threads do not need to communicate or coordinate their work in any way. To increase the degree of parallelism (DOP), SQL Server can simply add more threads and adjust the partitioning function. In practice, SQL Server uses a hash function to distribute rows for a hash aggregate. The hash function handles any data type, any number of GROUP BY columns, and any number of threads.

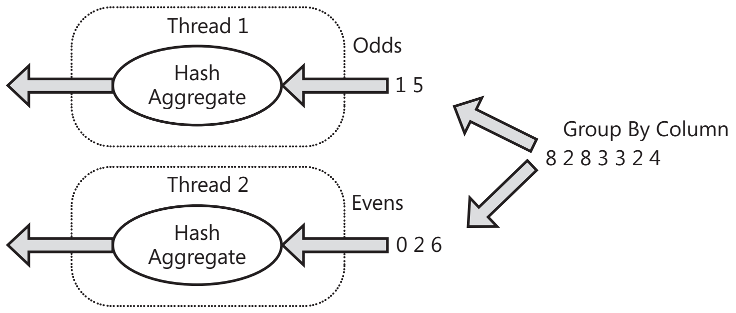


Figure 3-23 Executing an aggregate query on two threads

Note that this method of parallelism is not the same as "pipeline" parallelism in which multiple unrelated operators run concurrently in different threads. Although SQL Server frequently places different operators in different threads, the primary reason for doing so is to allow repartitioning of the data as it flows from one operator to the next. With pipeline parallelism, the degree of parallelism and the total number of threads would be limited to the number of operators.

The query optimizer decides whether to execute a query in parallel. For the optimizer even to consider a parallel plan, the following criteria must be met:

- SQL Server must be running on a multiprocessor, multicore, or hyperthreaded machine.
- The *affinity mask* (and *affinity mask64* for 64 processor servers) advanced configuration option must allow SQL Server to use at least two processors. The default setting of zero for affinity mask allows SQL Server to use all available processors.
- The *max degree of parallelism* advanced configuration option must be set to zero (the default) or to more than one. We discuss this option in more detail below.

Like most other decisions, the choice of whether to choose a serial or a parallel plan is cost-based. A complex and expensive query that processes many rows is more likely to result in a parallel plan than a simple query that processes very few rows. Although it is rarely necessary, you can also adjust the *cost threshold for parallelism* advanced configuration setting to raise or lower the threshold above which the optimizer considers parallel plans.

Degree of Parallelism (DOP)

The DOP is not part of the cached compiled plan and may change with each execution. SQL Server decides the DOP at the start of execution as follows:

1. If the query includes a MAXDOP N query hint, SQL Server sets the maximum DOP for the query to *N* or to the number of available processors (limited by the *affinity mask* configuration option) if *N* is zero.
2. If the query does not include a MAXDOP N query hint, SQL Server sets the maximum DOP to the setting of the *max degree of parallelism* advanced configuration option.

As with the MAXDOP N query hint, if this option is set to zero (the default), SQL Server sets the maximum DOP to the number of available processors.

3. SQL Server calculates the maximum number of concurrent threads that it needs to execute the query plan (as we will see momentarily, this number can exceed the DOP) and compares this result to the number of available threads. If there are not enough available threads, SQL Server reduces the DOP, as necessary. In the extreme case, SQL Server switches the parallel plan back to a serial plan. A serial plan runs with just the single connection thread and, thus, can always execute. Once the DOP is fixed, SQL Server reserves sufficient threads to ensure that the query can execute without running out of threads. This process is similar to how SQL Server acquires a memory grant for memory-consuming queries. The principle difference is that a query may wait to acquire a memory grant but never waits to reserve threads.

The *max worker threads* advanced configuration option determines the maximum number of threads available to SQL Server for system activities, as well as for parallel query execution. The default setting for this option varies depending on the number of processors. It is set higher for systems with more processors. It is also set higher for 64-bit servers than for 32-bit servers. It is rarely necessary to change the setting of this option.

As noted above, the number of threads used by a query may exceed the DOP. If you check *sys.dm_os_tasks* while running a parallel query, you may see more threads than the DOP. The number of threads may exceed the DOP because, if SQL Server needs to repartition data between two operators, it places them in different threads. The DOP only determines the number of threads per operator, not the total number of threads per query plan. As with the memory grant computation, when SQL Server computes the maximum number of threads that a query plan may consume, it does take blocking or stop-and-go operators into account. The operators above and below a blocking operator are never executed at the same time and, thus, can share both memory and threads.

In SQL Server 2000 if the DOP was less than the number of CPUs, the extra threads could use the extra CPUs, effectively defeating the MAXDOP settings. In SQL Server 2005, when a query runs with a given DOP, SQL Server also limits the number of schedulers used by that query to the selected DOP. That is, all threads used by the query are assigned to the same set of DOP schedulers, and the query uses only DOP CPUs, regardless of the total number of threads.

The Parallelism Operator (also known as Exchange)

The actual partitioning and movement of data between threads is handled by the parallelism (or exchange) iterator. Although it is unique in many respects, the parallelism iterator implements the same interfaces as any other iterator. Most of the other iterators do not need to be aware that they are executing in parallel. The optimizer simply places appropriate parallelism iterators in the plan and it runs in parallel.

The exchange iterator is unique in that it is really two iterators: a producer and a consumer. SQL Server places the producer at the root of a query subtree (often called a branch). The producer reads input rows from its subtree, assembles the rows into packets, and routes these packets to the appropriate consumer(s). SQL Server places the consumer at the leaf of the next query subtree. The consumer receives packets from its producer(s), removes the rows from these packets, and returns the rows to its parent iterator. For example, as Figure 3-24 illustrates, a repartition exchange running at DOP2 consists of two producers and two consumers:

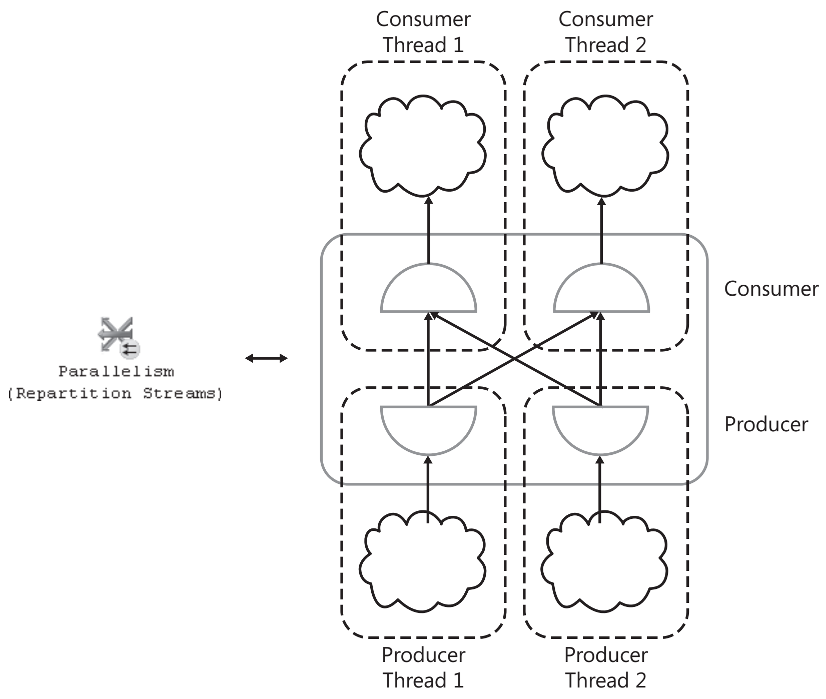


Figure 3-24 A repartition exchange running on two CPUs

Note that while the data flow between most iterators is pull-based (an iterator calls GetRow on its child when it is ready for another row), the data flow in between an exchange producer and consumer is push-based. That is, the producer fills a packet with rows and "pushes" it to the consumer. This model allows the producer and consumer threads to execute independently. SQL Server does have flow control to prevent a fast producer from flooding a slow consumer with excessive packets.

Exchanges can be classified in three different ways:

- by the number of producer and consumer threads,
- by the partitioning function used by the exchange to route rows from a producer thread to a consumer thread, and
- according to whether the exchange preserves the order of the input rows.

Table 3-6 shows how we classify exchanges based on the number of producer and/or consumer threads. The number of threads within each parallel "zone" of a query plan is the same and is equal to the DOP. Thus, an exchange may have exactly one producer or consumer thread if it is at the beginning or end of a serial zone within the plan or it may have exactly DOP producer or consumer threads.

Table 3-6 Types of Parallelism Exchange Operators

Type	# Producer Threads	# Consumer Threads
Gather Streams	DOP	1
Repartition Streams	DOP	DOP
Distribute Streams	1	DOP

A gather streams exchange is often called a *start parallelism* exchange because the operators above it run serially while the operators below it run in parallel. The root exchange in any parallel plan is always a gather exchange since the results of any query plan must ultimately be funneled back to the single connection thread to be returned to the client. A distribute streams exchange is often called a *stop parallelism* exchange. It is the opposite of a gather streams exchange. The operators above a distribute steams exchange run in parallel, while the operators below it run serially.

Table 3-7 shows how we classify exchanges based on the type of partitioning. Partitioning type only makes sense for a repartition or a distribute streams exchange. There is only one way to route rows in a gather exchange: to the single consumer thread.

Table 3-7 Types of Partitioning for Executing Parallel Queries

Partitioning Type	Description
Broadcast	Send all rows to all consumer threads.
Hash	Determine where to send each row by evaluating a hash function on one or more columns in the row.
Round Robin	Send each packet of rows to the next consumer thread in sequence.
Demand	Send the next row to the next consumer that asks for a row. This partition type is the only type of exchange that uses a pull rather than a push model for data flow. Demand partitioning is used only to distribute partition ids in parallel plans with partitioned tables.
Range	Determine where to send each row by evaluating a range function on one column in the row. Range partitioning is used only by certain parallel index build and statistics-gathering plans.

Finally, exchanges can be broken down into merging (or order preserving) and nonmerging (or non-order–preserving) exchanges. The consumer in a merging exchange ensures that rows from multiple producers are returned in a sorted order. (The rows must already be in this sorted order at the producer; the merging exchange does not actually sort.) A merging exchange only makes sense for a gather or a repartition streams exchange; with a distribute streams exchange, there is only one producer and, thus, only one stream of rows and nothing to merge at each consumer.

SQL Server commonly uses merging exchanges in plans for queries with ORDER BY clauses or in plans with parallel stream aggregate or merge join operators. However, merging exchanges are generally more expensive than, and do not scale as well as, nonmerging exchanges. A merging exchange cannot return rows until it has received rows from every input; even then, a merging exchange must returns rows in sorted order. If some inputs to a merging exchange include rows that are much later in the sort order than rows from the other inputs, the exchange cannot return the rows that sort later. The inputs returning the rows that sort later may get too far ahead of the other inputs and be forced to wait while the other inputs "catch up." The higher the DOP (that is, the more threads), the more likely that some of the inputs will be forced to wait for other inputs.

In some extreme cases, merging exchanges may lead to intra-query "parallel deadlocks." For a query plan to be susceptible to a parallel deadlock, it must include at least two merging exchanges separated by order-preserving operators or a parallel merge join above a pair of merging exchanges. Although parallel deadlocks do not cause a query to fail, they can lead to serious performance degradations. Parallel deadlocks were a far more common problem in SQL Server 2000 than in SQL Server 2005. Improvements in SQL Server 2005 have resolved this problem in all but a few very rare scenarios. To check for a parallel deadlock, use the *sys.dm_os_waiting_tasks* DMV. If all threads associated with a session are blocked on the CXPACKET wait type, you have a parallel deadlock. Generally, the only solution to a parallel deadlock is to reduce the DOP, force a serial plan, or alter the query plan to eliminate the merging exchanges.

In light of these performance issues, the optimizer costs parallel plans with merging exchanges fairly high and tries to avoid choosing a plan with merging exchanges as the DOP increases.

SQL Server includes all of the above properties in all three query plan varieties (graphical, text, and XML). Moreover, in graphical query plans you can also tell at a glance which operators are running in parallel (that is, which operators are between a start exchange and a stop exchange) by looking for a little parallelism symbol on the operator icons, as illustrated in Figure 3-25.

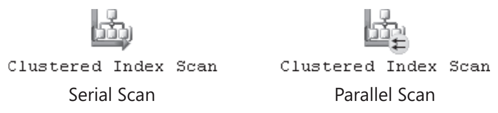


Figure 3-25 Clustered Index Scan operator icon with and without parallelism

In the remainder of this subsection, we will look at how a few of the operators that we've already discussed earlier in this chapter behave in parallel plans. Since most operators neither need to know nor care whether they are executing in parallel and can be parallelized merely by placing them below a start exchange, we will focus primarily on the handful of operators whose behavior is interesting in the context of parallelism.

Parallel Scan

The scan operator is one of the few operators that is parallel "aware." The threads that compose a parallel scan work together to scan all of the rows in a table. There is no a priori assignment of rows or pages to a particular thread. Instead, the storage engine dynamically hands out pages or ranges of rows to threads. A parallel page supplier coordinates access to the pages or rows of the table. The parallel page supplier ensures that each page or range of rows is assigned to exactly one thread and, thus, is processed exactly once.

At the beginning of a parallel scan, each thread requests a set of pages or a range of rows from the parallel page supplier. The threads then begin processing their assigned pages or rows and begin returning results. When a thread finishes with its assigned set of pages, it requests the next set of pages or the next range of rows from the parallel page supplier.

This algorithm has a couple of advantages:

1. It is independent of the number of threads. SQL Server can add and remove threads from a parallel scan, and it automatically adjusts. If the number of threads doubles, each thread processes (approximately) half as many pages. And, if the I/O system can keep up, the scan runs twice as fast.
2. It is resilient to skew or load imbalances. If one thread runs slower than the other threads, that thread simply requests fewer pages while the other faster threads pick up the extra work. The total execution time degrades smoothly. Compare this scenario to what would happen if SQL Server statically assigned pages to threads: The slow thread would dominate the total execution time.

Let's begin with a simple example. To get parallel plans, we need fairly big tables; if the tables are too small, the optimizer concludes that a serial plan is perfectly adequate. The following script creates two tables. Each table has 250,000 rows and, thanks to the fixed-length *char(200)* column, well over 6,500 pages.

```
CREATE TABLE [HugeTable1]
   (
   [Key] int,
   [Data] int,
   [Pad] char(200),
   CONSTRAINT [PK1] PRIMARY KEY ([Key])
   )

SET NOCOUNT ON
DECLARE @i int
BEGIN TRAN
SET @i = 0
WHILE @i < 250000
```

```
BEGIN
   INSERT [HugeTable1] VALUES(@i, @i, NULL)
   SET @i = @i + 1
   IF @i % 1000 = 0
   BEGIN
      COMMIT TRAN
      BEGIN TRAN
   END
END
COMMIT TRAN

SELECT [Key], [Data], [Pad]
INTO [HugeTable2]
FROM [HugeTable1]

ALTER TABLE [HugeTable2]
ADD CONSTRAINT [PK2] PRIMARY KEY ([Key])
```

Now let's try the simplest possible query:

```
SELECT [Key], [Data]
FROM [HugeTable1]
```

Despite the large table, this query results in a serial plan:

```
|--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]))
```

We do not get a parallel plan because parallelism is really about speeding up queries by applying more CPUs to the problem. The cost of this query is dominated by the cost of reading pages from disk (which is mitigated by readahead rather than parallelism) and returning rows to the client. The query uses relatively few CPU cycles and, in fact, would probably run slower if SQL Server parallelized it.

Now suppose we add a fairly selective predicate to the query:

```
SELECT [Key], [Data]
FROM [HugeTable1]
WHERE [Data] < 1000
```

This query results in a parallel plan:

```
|--Parallelism(Gather Streams)
   |--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]),
       WHERE:([HugeTable1].[Data]<CONVERT_IMPLICIT(int,[@1],0)))
```

Since we do not have an index on the *Data* column, SQL Server cannot perform an index seek and must evaluate the predicate for each row. By running this query in parallel, SQL Server distributes the cost of evaluating the predicate across multiple CPUs. (In this case, the predicate is so cheap that it probably does not make much difference whether or not the query runs in parallel.)

Note that the exchange in this plan does not return the rows in any particular order. It simply returns the rows in whatever order it receives them from its producer threads. Now observe what happens if we add an ORDER BY clause to the query:

```
SELECT [Key], [Data]
FROM [HugeTable1]
WHERE [Data] < 1000
ORDER BY [Key]
```

The optimizer recognizes that we have a clustered index that can return rows sorted by the *Key* column. To exploit this index and avoid an explicit sort, the optimizer adds a merging or order-preserving exchange to the plan:

```
|--Parallelism(Gather Streams, ORDER BY:([HugeTable1].[Key] ASC))
   |--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]),
      WHERE:([HugeTable1].[Data]<CONVERT_IMPLICIT(int,[@1],0)) ORDERED FORWARD)
```

We can identify the merging exchange by the ORDER BY clause in the query plan. This clause indicates the order in which the exchange returns the rows.

Load Balancing As we noted previously, the parallel scan algorithm dynamically allocates pages to threads. We can devise an experiment to see this effect in action. Consider this query:

```
SELECT MIN([Data]) FROM [HugeTable1]
```

This query scans the entire table, but because of the aggregate it uses a parallel plan. The aggregate also ensures that the query returns only one row. Without it, the execution time of this query would be dominated by the cost of returning rows to the client. This overhead could alter the results of our experiment.

```
|--Stream Aggregate(DEFINE:([Expr1003]=MIN([partialagg1004])))
   |--Parallelism(Gather Streams)
      |--Stream Aggregate(DEFINE:([partialagg1004]=MIN([HugeTable1].[Data])))
         |--Clustered Index Scan(OBJECT:([HugeTable1].[PK1]))
```

Before proceeding with the experiment, notice that this plan includes two stream aggregates instead of the usual one in a serial plan. We refer to the bottommost aggregate as a local or partial aggregate. The local aggregate computes the minimum value of the *Data* column within each thread. We refer to the topmost aggregate as a global aggregate. The global aggregate computes the minimum of the local minimums. Local aggregation improves query performance in two ways. First, it reduces the number of rows flowing through the exchange. Second, it enables parallel execution of the aggregate. Since this query computes a single result row (recall that it is a scalar aggregate), without the local aggregate, all of the scanned rows would have to be processed by a single thread. SQL Server can use local aggregation for both stream and hash aggregates, for most built-in and many user-defined aggregate functions, and for both scalar aggregates (as in this example) and grouping aggregates. In the case of grouping aggregates, the benefits of local aggregation increase as the number of groups decreases. The smaller the number of groups, the more work the local aggregate is actually able to perform.

Returning to our experiment, using SET STATISTICS XML ON we can run the above query and see exactly how many rows each thread processes. (The same information is also visible with graphical showplan in the Properties window.) Here is an excerpt of the XML output on a two-processor system:

```
<RelOp NodeId=“3” PhysicalOp=“Clustered Index Scan” LogicalOp=“Clustered Index Scan”...>
  <RunTimeInformation>
    <RunTimeCountersPerThread Thread=“1” ActualRows=“124986”... />
    <RunTimeCountersPerThread Thread=“2” ActualRows=“125014”... />
    <RunTimeCountersPerThread Thread=“0” ActualRows=“0”... />
  </RunTimeInformation>
</RelOp>
```

We can see that both threads (threads 1 and 2) processed approximately half of the rows. (Thread 0 is the coordinator or main thread. It only executes the portion of the query plan above the topmost exchange. Thus, we do not expect it to process any rows for operators executed in a parallel portion of the query plan.)

Now let’s repeat the experiment, but this time let’s run an expensive serial query at the same time. This cross join query will run for a very long time and use plenty of CPU cycles. The OPTION (MAXDOP 1) query hint forces SQL Server to execute this query in serial. We will discuss this and other hints in more detail in Chapters 4 and 5.

```
SELECT MIN(T1.[Key] + T2.[Key])
FROM [HugeTable1] T1 CROSS JOIN [HugeTable2] T2
OPTION (MAXDOP 1)
```

This serial query runs with a single thread and consumes cycles from only one of the two schedulers. While it is running, we run the parallel scan query again. Here is the statistics XML output for the second run:

```
<RelOp NodeId=“3” PhysicalOp=“Clustered Index Scan” LogicalOp=“Clustered Index Scan”...>
  <RunTimeInformation>
    <RunTimeCountersPerThread Thread=“1” ActualRows=“54232”... />
    <RunTimeCountersPerThread Thread=“2” ActualRows=“195768”... />
    <RunTimeCountersPerThread Thread=“0” ActualRows=“0”... />
  </RunTimeInformation>
</RelOp>
```

This time thread 1 processed over 75 percent of the rows while thread 2, which was busy executing the serial plan, processed fewer than 25 percent of the rows. The parallel scan automatically balanced the work across the two threads. Because thread 1 had more free cycles (it wasn’t competing with the serial plan), it requested and scanned more pages.

Caution If you try this experiment, don’t forget to terminate the serial query when you are done! Otherwise, it will continue to run and waste cycles for a very long time.

The same load balancing that we just observed applies equally whether a thread is slowed down because of an external factor (such as the serial query in this example) or because of an internal factor. For example, if it costs more to process some rows than others, we will see the same behavior.

Parallel Nested Loops Join

SQL Server parallelizes a nested loops join by distributing the outer rows (that is, the rows from the first input) randomly among the nested loops join threads. For example, if there are two threads running a nested loops join, SQL Server sends about half of the rows to each thread. Each thread then runs the inner side (that is, the second input) of the nested loops join for its set of rows as if it were running serially. That is, for each outer row assigned to it, the thread executes its inner input using that row as the source of any correlated parameters. In this way, the threads can run independently. SQL Server does not add exchanges to or parallelize the inner side of a nested loops join.

Here is a simple example of a parallel nested loops join:

```
SELECT T1.[Key], T1.[Data], T2.[Data]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
   ON T1.[Key] = T2.[Key]
WHERE T1.[Data] < 100
```

Let's analyze the STATISTICS PROFILE output for this plan:

```
Rows Executes
100   1   |--Parallelism(Gather Streams)
100   2      |--Nested Loops(Inner Join, OUTER REFERENCES:([T1].[Key]) OPTIMIZED)
100   2         |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]),
                   WHERE:([T1].[Data]<(100)))
100   100        |--Clustered Index Seek (OBJECT:([HugeTable2].[PK2] AS [T2]),
                    SEEK:([T2].[Key]=[T1].[Key])
                    ORDERED FORWARD)
```

Note that there is only one exchange (in other words, parallelism operator) in this plan. This exchange is at the root of the query plan, therefore all of the operators in this plan (the nested loops join, the table scan, and the clustered index seek) execute in each thread. The lack of an index on the *Data* column of *HugeTable1* forces the clustered index scan and the use of a residual predicate to evaluate T1.[Data] < 100. Because there are 250,000 rows in *HugeTable1*, the scan is expensive and, as in the previous examples, the optimizer chooses a parallel scan of *HugeTable1*. The scan of *HugeTable1* is *not* immediately below an exchange (in other words, a parallelism operator). In fact, it is on the outer side of the nested loops join, and it is the nested loops join that is below the exchange. Nevertheless, because the scan is on the outer side of the join and because the join is below a start (in other words, a gather) exchange, SQL Server performs a parallel scan of *HugeTable1*. Since a parallel scan assigns pages to threads dynamically, the scan distributes the *HugeTable1* rows among the threads. It does not mater which rows it distributes to which threads.

Because SQL Server ran this query with DOP 2, we see that there are two executes each for the clustered index scan of *HugeTable1* and for the join (both of which are in the same thread). Moreover, the scan and join both return a total of 100 rows, though we cannot tell from this output how many rows each thread returned. We can (and momentarily will) determine this information using statistics XML output.

Next, the join executes its inner side (in this case, the clustered index seek on *HugeTable2*) for each of the 100 outer rows. Here is where things get a little tricky. Although each of the two threads contains an instance of the clustered index seek iterator, and even though the seek is below the join, which is below the exchange, the seek is on the *inner* side of the join and, thus, the seek does *not* use a parallel scan. Instead, the two seek instances execute independently of one another on two different outer rows and two different correlated parameters. As in a serial plan, we see 100 executes of the index seek: one for each row on the outer side of the join. With only one exception, which we will address below, no matter how complex the inner side of a nested loops join is, we always execute it as a serial plan just as in this simple example.

Round-Robin Exchange In the previous example, SQL Server relies on the parallel scan to distribute rows uniformly among the threads. In some cases, SQL Server must add a round-robin exchange to distribute the rows. (Recall that a round-robin exchange sends each subsequent packet of rows to the next consumer thread in a fixed sequence.) Here is one such example:

```
SELECT T1_Top.[Key], T1_Top.[Data], T2.[Data]
FROM
   (
   SELECT TOP 100 T1.[Key], T1.[Data]
   FROM [HugeTable1] T1
   ORDER BY T1.[Data]
   ) T1_Top,
   [HugeTable2] T2
WHERE T1_Top.[Key] = T2.[Key]
```

Here is the corresponding plan:

```
|--Parallelism(Gather Streams)
   |--Nested Loops(Inner Join, OUTER REFERENCES:([T1].[Key],
      [Expr1004]) WITH UNORDERED PREFETCH)
      |--Parallelism(Distribute Streams, RoundRobin Partitioning)
      |    |--Top(TOP EXPRESSION:((100)))
      |       |--Parallelism(Gather Streams, ORDER BY:([T1].[Data] ASC))
      |          |--Sort(TOP 100, ORDER BY:([T1].[Data] ASC))
      |             |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]))
      |--Clustered Index Seek(OBJECT:([HugeTable2].[PK2] AS [T2]),
         SEEK:([T2].[Key]=[T1].[Key]) ORDERED FORWARD)
```

The main difference between this plan and the plan from the original example is that this plan includes a top-100 rows iterator. The top iterator can only be correctly evaluated in a serial plan thread. (It cannot be split among multiple threads or we might end up with too few or too many rows.) Thus, SQL Server must add a stop (that is, a distribute streams) exchange above the top iterator and cannot use the parallel scan to distribute the rows among the join threads. Instead SQL Server parallelizes the join by having the stop exchange use round-robin partitioning to distribute the rows among the join threads.

Parallel Nested Loops Join Performance The parallel scan has one major advantage over the round-robin exchange. A parallel scan automatically and dynamically balances the workload among the threads, while a round-robin exchange does not. As the previous parallel scan example demonstrates, if we have a query in which one thread is slower than the others, the parallel scan may help compensate.

Both the parallel scan and a round-robin exchange may fail to keep all join threads busy if there are too many threads and too few pages and/or rows to be processed. Some threads may get no rows to process and end up idle. This problem can be more pronounced with a parallel scan since it doles out multiple pages at one time to each thread while the exchange distributes one packet (equivalent to one page) of rows at one time.

We can see this problem in the original parallel nested loops join example above by checking the statistics XML output:

```
<RelOp NodeId="1" PhysicalOp="Nested Loops" LogicalOp="Inner Join"...>
  <RunTimeInformation>
    <RunTimeCountersPerThread Thread="2" ActualRows="0"... />
    <RunTimeCountersPerThread Thread="1" ActualRows="100"... />
    <RunTimeCountersPerThread Thread="0" ActualRows="0"... />
  </RunTimeInformation>
</RelOp>
```

This output shows that all of the join's output rows are processed by thread 1. The problem is that the clustered index scan of *HugeTable1* has a residual predicate T1.[Data] < 100, which is true for the first 100 rows in the table and false for the remaining rows. The roughly three pages containing the first 100 rows are all assigned to the first thread.

In this example, this is not a big problem since the inner side of the join is fairly cheap and contributes a small percentage of the overall cost of the query (with the clustered index scan of *HugeTable1* itself contributing a much larger percentage of the cost). However, this problem could be more significant if the inner side of the query were more expensive. The problem is especially notable in SQL Server 2005 with partitioned tables which, as noted earlier, use nested loops joins to enumerate partition ids. For instance, if we try to perform a parallel scan of two partitions of a large table, the DOP of the scan is effectively limited to two because of the nested loops join.

Inner-side Parallel Execution We noted above that, with one exception, SQL Server always executes the inner side of a parallel nested loops join as a serial plan. The exception occurs when the optimizer knows that the outer input to the nested loops join is guaranteed to return only a single row and when the join has no correlated parameters. That is, the inner side of the join is guaranteed to run exactly once and can be run independently of the outer side of the join. In this case, if the join is an inner join, the inner side of the join may include a parallel scan. Moreover, in some rare scenarios, if the join is an outer or semi-join, it may be executed in a serial zone (below a stop exchange) and the inner side of the join may include exchanges.

For example, consider this query:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
   ON T1.[Data] = T2.[Data]
WHERE T1.[Key] = 0
```

This query yields the following plan:

```
|--Parallelism(Gather Streams)
   |--Nested Loops(Inner Join, WHERE:([T1].[Data]=[T2].[Data]))
      |--Parallelism(Distribute Streams, Broadcast Partitioning)
      |    |--Clustered Index Seek (OBJECT:([HugeTable1].[PK1] AS [T1]),
                SEEK:([T1].[Key]=(0)) ORDERED FORWARD)
      |--Clustered Index Scan(OBJECT:([HugeTable2].[PK2] AS [T2]))
```

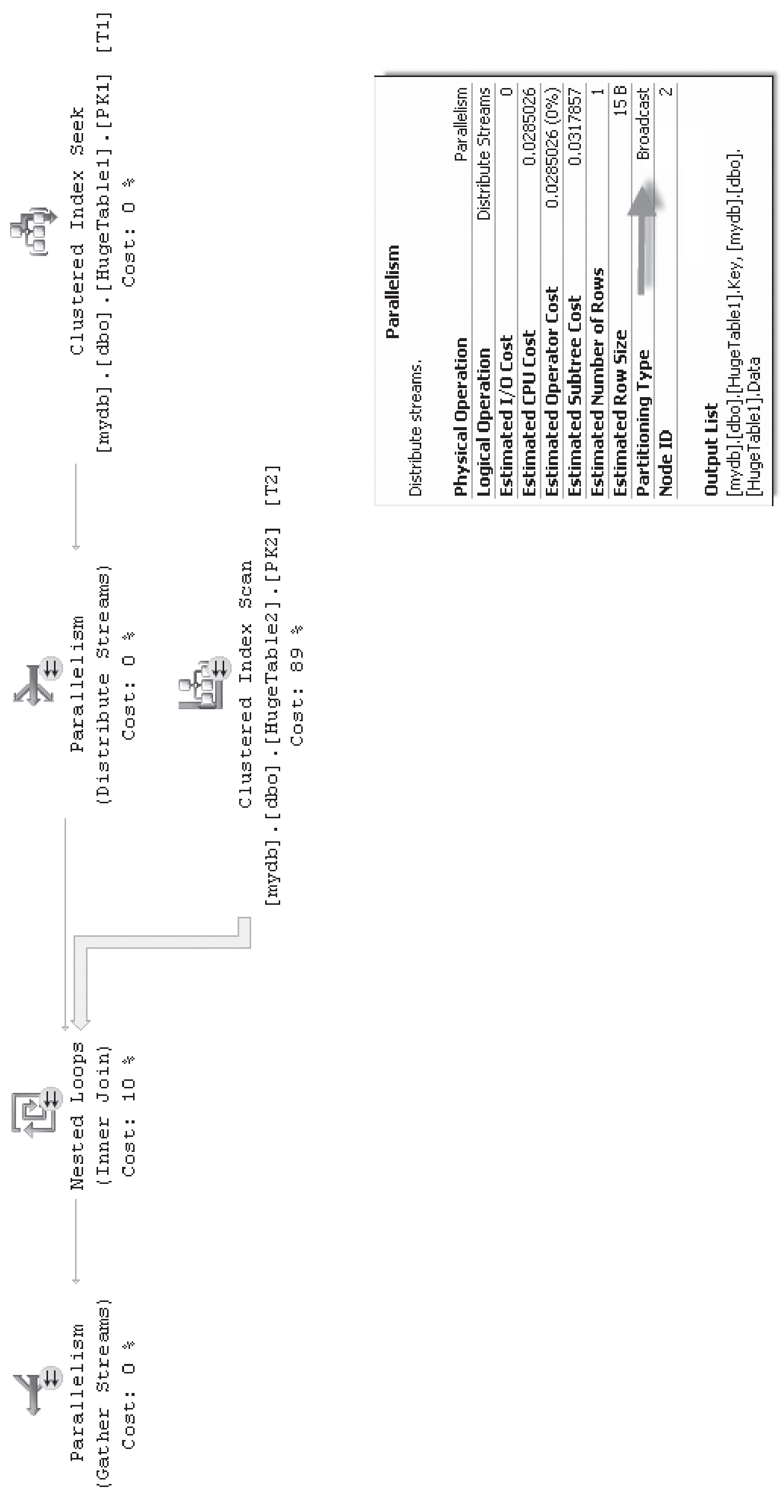
The equality predicate T1.[Key] = 0 ensures that the clustered index seek of *HugeTable1* returns exactly one row. Notice that the plan includes a broadcast exchange, which delivers this single row to each instance of the nested loops join. In this plan, the clustered index scan of *HugeTable2* uses a parallel scan. That is, all instances of this scan cooperatively scan *HugeTable2* exactly once. If this plan used a serial scan of *HugeTable2*, each scan instance would return the entire contents of *HugeTable2* which, in conjunction with the broadcast exchange, would result in the plan returning each row multiple times. Figure 3-26 shows the graphical plan for this query, along with the ToolTip indicating that a broadcast exchange is being used.

Parallel Merge Join

SQL Server parallelizes merge joins by distributing both sets of input rows among the individual merge join threads using hash partitioning. If two input rows join, they have the same values for the join key and, therefore, hash to the same merge join thread. Unlike the parallel plan examples we've seen so far, merge join also requires that its input rows be sorted. In a parallel merge join plan, as in a serial merge join plan, SQL Server can use an index scan to deliver rows to the merge join in the correct sort order. However, in a parallel plan, SQL Server must also use a merging exchange to preserve the order of the input rows.

Although, as we discussed previously, the optimizer tends to favor plans that do not require a merging exchange; by including an ORDER BY clause in a join query, we can encourage such a plan. For example, consider the following query:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
   ON T1.[Key] = T2.[Data]
ORDER BY T1.[Key]
```



Parallelism

Distribute streams.

Physical Operation	Parallelism
Logical Operation	Distribute Streams
Estimated I/O Cost	0
Estimated CPU Cost	0.0285026
Estimated Operator Cost	0.0285026 (0%)
Estimated Subtree Cost	0.0317857
Estimated Number of Rows	1
Estimated Row Size	15 B
Partitioning Type	Broadcast
Node ID	2

Output List
[mydb].[dbo].[HugeTable1].Key, [mydb].[dbo].[HugeTable1].Data

Figure 3-26 A parallel plan using a broadcast exchange

The optimizer chooses the following query plan:

```
|--Parallelism(Gather Streams, ORDER BY:([T1].[Key] ASC))
   |--Merge Join(Inner Join, MERGE:([T1].[Key])=([T2].[Data]), RESIDUAL:(...))
      |--Parallelism(Repartition Streams, Hash Partitioning,
         PARTITION COLUMNS:([T1].[Key]), ORDER BY:([T1].[Key] ASC))
      |  |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]), ORDERED FORWARD)
      |--Sort(ORDER BY:([T2].[Data] ASC))
         |--Parallelism(Repartition Streams, Hash Partitioning,
           PARTITION COLUMNS:([T2].[Data]))
      |  |--Clustered Index Scan (OBJECT:([HugeTable2].[PK2] AS [T2]))
```

This plan includes three exchanges including two merging exchanges. Two of the exchanges are hash partitioning exchanges and are placed below the merge join. These exchanges ensure that any input rows that might potentially join are delivered to the same merge join thread. The first merge join input uses a merging exchange, and the clustered index scan of *HugeTable1* delivers the input rows sorted on the join key. The second merge join input uses a nonmerging exchange. The clustered index scan of *HugeTable2* does not deliver input rows sorted on the join key. Instead there is a sort between the merge join and the exchange. By placing the exchange below the sort, the optimizer avoids the need to use a merging exchange. The final exchange is the start exchange at the top of the plan. This exchange returns the rows in the order required by the ORDER BY clause on the query and takes advantage of the property that the merge join output rows in the same order as it inputs them.

Parallel Hash Join

SQL Server uses one of two different strategies to parallelize a hash join. One strategy uses hash partitioning just like a parallel merge join; the other strategy uses broadcast partitioning and is often called a broadcast hash join.

Hash Partitioning The more common strategy for parallelizing a hash join involves distributing the build rows (in other words, the rows from the first input) and the probe rows (in other words, the rows from the second input) among the individual hash join threads using hash partitioning. As in the parallel merge join case, if two input rows join, they are guaranteed to hash to the same hash join thread. After the data has been hash partitioned among the threads, the hash join instances all run completely independently on their respective data sets. Unlike merge join, hash join does not require that input rows be delivered in any particular order and, thus, it does not require merging exchanges. The absence of any interthread dependencies ensures that this strategy scales extremely well as the DOP increases.

To see an example of a parallel hash join, consider the following simple query:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
  ON T1.[Data] = T2.[Data]
```

SQL Server executes this query using the following plan:

```
|--Parallelism(Gather Streams)
   |--Hash Match(Inner Join, HASH:([T1].[Data])=([T2].[Data]), RESIDUAL:(...))
      |--Parallelism(Repartition Streams, Hash Partitioning,
         PARTITION COLUMNS:([T1].[Data]))
      |  |--Clustered Index Scan (OBJECT:([HugeTable1].[PK1] AS [T1]))
      |--Parallelism(Repartition Streams, Hash Partitioning,
         PARTITION COLUMNS:([T2].[Data]))
      |  |--Clustered Index Scan (OBJECT:([HugeTable2].[PK2] AS [T2]))
```

Broadcast Partitioning Consider what happens if SQL Server tries to parallelize a hash join using hash partitioning, but there are only a small number of rows on the build side of the hash join. If there are fewer rows than hash join threads, some threads might receive no rows at all. In this case, those threads would have no work to do during the probe phase of the join and would remain idle. Even if there are more rows than threads, the presence of duplicate key values and/or skew in the hash function means some threads might receive and process many more rows than other threads.

To reduce the risk of skew problems, when the optimizer estimates that the number of build rows is relatively small, it may choose to broadcast these rows to all of the hash join threads. Since all build rows are broadcast to all hash join threads, in a broadcast hash join, it does not matter which threads process which probe rows. Each probe row can be sent to any thread and, if it can join with any build rows, it will.

For example, the following query includes a very selective predicate:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
   ON T1.[Data] = T2.[Data]
WHERE T1.[Key] < 100
```

Because of the selective predicate, the optimizer chooses a broadcast hash join plan:

```
|--Parallelism(Gather Streams)
   |--Hash Match(Inner Join, HASH:([T1].[Data])=([T2].[Data]), RESIDUAL:(...))
      |--Parallelism(Distribute Streams, Broadcast Partitioning)
      |   |--Clustered Index Seek (OBJECT:([HugeTable1].[PK1] AS [T1]),
              SEEK:([T1].[Key] < (100)) ORDERED FORWARD)
      |--Clustered Index Scan(OBJECT:([HugeTable2].[PK2] AS [T2]))
```

Note that the exchange above the clustered index seek of *HugeTable1* is now a broadcast exchange, while the exchange above the clustered index scan of *HugeTable2* is gone. This plan does not need an exchange above the scan of *HugeTable2* because the parallel scan automatically distributes the pages and rows of *HugeTable2* among the hash join threads. This result is similar to how the parallel scan distributed rows among nested loops join threads for the parallel nested loops join, which we discussed earlier. Similar to the parallel nested loops join, if there is a serial zone on the probe input of a broadcast hash join (for example, caused by a top operator), the query may use a round-robin exchange to redistribute the rows.

While broadcast hash joins do reduce the risk of skew problems, they are not suitable for all scenarios. In particular, broadcast hash joins use more memory than their hash-partitioned counterparts. Because a broadcast hash join means SQL Server sends every build row to every hash join thread, if the number of threads doubles, the amount of memory consumed also doubles. Thus, a broadcast hash join requires memory that is proportional to the degree of parallelism, whereas a hash-partitioned parallel hash join requires the same amount of memory regardless of the degree of parallelism.

Bitmap Filtering Next, suppose we have a moderately selective predicate on the build input to a hash join:

```
SELECT T1.[Key], T1.[Data], T2.[Key]
FROM [HugeTable1] T1 JOIN [HugeTable2] T2
   ON T1.[Data] = T2.[Data]
WHERE T1.[Key] < 10000
```

The predicate T1.[Key] < 10000 is not selective enough to generate a broadcast hash join. This predicate does, however, eliminate 96 percent of the build rows from *HugeTable1*. It also indirectly eliminates 96 percent of the rows from *HugeTable2* because they no longer join with rows from *HugeTable1*. It would be nice if there was a way to eliminate these rows from *HugeTable2* much earlier without the overhead of passing the rows through the exchange and into the hash join. The bitmap operator provides just such a mechanism:

```
|--Parallelism(Gather Streams)
   |--Hash Match(Inner Join, HASH:([T1].[Data])=([T2].[Data]), RESIDUAL:(...))
      |--Bitmap(HASH:([T1].[Data]), DEFINE:([Bitmap1004]))
      |  |--Parallelism(Repartition Streams, Hash Partitioning,
             PARTITION COLUMNS:([T1].[Data]))
      |    |--Clustered Index Seek (OBJECT:([HugeTable1].[PK1] AS [T1]),
               SEEK:([T1].[Key] < (10000)) ORDERED FORWARD)
      |    |--Parallelism(Repartition Streams, Hash Partitioning,
               PARTITION COLUMNS:([T2].[Data]), WHERE:(PROBE([Bitmap1004])=TRUE))
      |    |--Clustered Index Scan (OBJECT:([HugeTable2].[PK2] AS [T2]))
```

As its name suggests, the bitmap operator builds a bitmap. Just like the hash join, the bitmap operator hashes each row of *HugeTable1* on the join key and sets the corresponding bit in the bitmap. Once the scan of *HugeTable1* and the hash join build are complete, SQL Server transfers the bitmap to the exchange above *HugeTable2*. This exchange uses the bitmap as a filter; notice the WHERE clause in the plan. The exchange hashes each row of *HugeTable2* on the join key and tests the corresponding bit in the bitmap. If the bit is set, the row may join and the exchange passes it along to the hash join. If the bit is not set, the row cannot join and the exchange discards it.

Although it is somewhat rarer, some parallel merge joins also use a bitmap operator. To use the bitmap operator, a merge join must have a sort on its left input. The optimizer adds the bitmap operator below the sort. Recall that a merge join processes both inputs concurrently. Without the sort, there would be no way to build the bitmap on one input before beginning to process the other input. However, because the sort is blocking, the plan can build a bitmap on the left input before the merge join begins processing its right input.

Inserts, Updates, and Deletes

Data Modification (INSERT, UPDATE, and DELETE) statements could be the subject of an entire chapter. In this section, we provide just a brief overview of data modification statement plans. These plans consist of two sections: a "read cursor" and a "write cursor." The read cursor determines which rows will be affected by the data modification statement. The read cursor works like a normal SELECT statement. All of the material about understanding queries that we have discussed in this chapter, including the discussion of parallelism, applies equally well to the read cursor of a data modification plan. The write cursor executes the actual INSERTS, UPDATES, or DELETES. The write cursor, which can get extremely complex, also executes other side effects of the data modification such as nonclustered index maintenance, indexed view maintenance, referential integrity constraint validation, and cascading actions. The write cursor is implemented using many of the same operators that we've already discussed. However, unlike the read cursor, where we can often impact performance by rewriting the query or by using hints (which we will discuss in Chapters 4 and 5), we have comparatively little control over the write cursor, query plan and performance. Short of creating or dropping indexes or constraints, we cannot control the list of indexes that must be maintained and constraints that must be validated. It is also worth noting that the write cursor is never executed using parallelism.

For example, consider the following UPDATE statement, which changes the shipper for orders shipped to London. This statement is guaranteed to violate the foreign key constraint on the *ShipVia* column and will fail.

```
UPDATE [Orders]
SET [ShipVia] = 4
WHERE [ShipCity] = N'London'
```

This statement uses the following query plan:

```
|--Assert(WHERE:(CASE WHEN [Expr1023] IS NULL THEN (0) ELSE NULL END))
   |--Nested Loops(Left Semi Join, OUTER REFERENCES:([Orders].[ShipVia]),
       DEFINE:([Expr1023] = [PROBE VALUE]))
       |--Clustered Index Update(OBJECT:([Orders].[PK_Orders]),
          OBJECT:([Orders].[ShippersOrders]),
          SET:([Orders].[ShipVia] = [Expr1019]))
       |   |--Compute Scalar(DEFINE:([Expr1021]=[Expr1021]))
       |      |--Compute Scalar(DEFINE:([Expr1021]=CASE WHEN [Expr1003]
                                THEN (1)
                                ELSE (0) END))
       |         |--Compute Scalar(DEFINE:([Expr1019]=(4)))
       |            |--Compute Scalar(DEFINE:([Expr1003]=
                         CASE WHEN [Orders].[ShipVia] = (4)
                         THEN (1) ELSE (0) END))
       |                 |--Top(ROWCOUNT est 0)
|                    |--Clustered Index Scan
                         (OBJECT:([Orders].[PK_Orders]),
                         WHERE:([Orders].[ShipCity]=N'London') ORDERED)
|--Clustered Index Seek(OBJECT:([Shippers].[PK_Shippers]),
  SEEK:([Shippers].[ShipperID]=[Orders].[ShipVia]) ORDERED FORWARD)
```

The read cursor for this plan consists solely of the clustered index scan of the *Orders* table, whereas the write cursor consists of the entire remainder of the plan. The write cursor includes a clustered index update operator, which updates two indexes on the *Orders* table—the *PK_Orders* clustered index and the *ShippersOrders* nonclustered index. The write cursor also includes a join with the *Shippers* table and an assert operator, which validate the foreign key constraint on the *ShipVia* column.

Summary

In this chapter, we've explained the basics of query processing in SQL Server. We've introduced iterators—the fundamental building blocks of query processing—and discussed how to view query plans in three different formats: text, graphical, and XML. We've covered many of the most common iterators, including the iterators that implement scans and seeks, joins, and aggregation. We've looked at more complex query plans including dynamic index seeks, index unions and intersection, and a variety of subquery examples. We've also explored how the query processor implements parallel query execution. All of the query plans that SQL Server supports are built from these and other basic operators.

The vast majority of the time, the SQL Server query processor works extremely well. The optimizer generally chooses satisfactory query plans, and most applications work well without any special tuning. However, there will be times when your queries are not performing optimally and in the next chapter we'll discuss options for detecting and correcting problems. The more you understand about what actually happens during query execution, the better you'll be able to detect when a plan is not optimal, and when there is room for improvement.