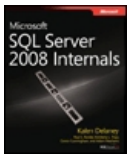


Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.

Chapter 8: The Query Optimizer

Conor Cunningham

The Query Optimizer in Microsoft SQL Server 2008 determines the query plan to be executed for a given SQL statement. Because the Query Optimizer does not have a lot of exposed features, it is not as widely understood as some of the other components in the SQL Server Engine. This chapter describes the Query Optimizer and how it works. After reading this chapter, you should understand the high-level optimizer architecture and should be able to reason about why a particular plan was selected by the Query Optimizer. By extension, you should be able to troubleshoot problem query plans in the case when the Query Optimizer may not select the desired query plan and how to affect that selection.

This chapter is split into two sections. The first section explains the basic mechanisms of the Query Optimizer. This includes the high-level structures that are used and how this defines the set of alternatives considered for each plan. The second section discusses specific areas in the Query Optimizer and how they fit into this framework. For example, it discusses topics like “How do indexes get selected?”, “How do statistics get used?”, and “How do I understand update plans?”

Overview

The basic compilation “pipeline” for a single query appears in Figure 8-1.

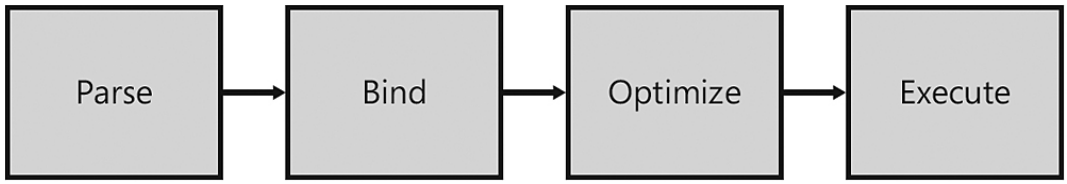


Figure 8-1: Query processor pipeline

When a query is compiled, the SQL statement is first *parsed* into an equivalent tree representation. For queries with valid SQL syntax, the next stage performs a series of validation steps on the query, generally called *binding,* where the columns and tables in the tree are compared to database metadata to make sure that those columns and tables exist and are visible to the current user. This stage also performs semantic checks on the query to make sure it is valid, such as making sure that the columns bound to a *GROUP BY* operation are valid. Once the query tree has been bound and is determined to be a valid query, the Query Optimizer takes the query and starts evaluating different possible query plans. The Query Optimizer performs this search and then selects the query plan to be executed and returns it to the system to execute. The *execution* component runs the query plan and returns the results of the query.

The SQL Server 2008 Query Optimizer has a number of additional features that extend this diagram to make it more useful for database developers and DBAs. For example, query plans are cached because they are expensive to produce and are often used repeatedly. Old query plans are recompiled if the underlying data has changed sufficiently. SQL Server also supports the T-SQL language, which means that batches of multiple statements can be processed in one request to the SQL Server Engine. The Query Optimizer does not consider batch compilation or workload analysis, so this chapter focuses on what happens in a single query’s compilation.

Tree Format

When you submit a SQL query to the query processor, the SQL string is parsed into a tree representation. Each node in the tree represents a query operation to be performed. For example, each table in the FROM clause has its own operator. A WHERE clause is also represented in a separate operator. Joins are represented with operators that have one input for each table. For example, the query *SELECT * FROM Customers C INNER JOIN Orders O on C.cid = O.cid WHERE O.date = ‘2008-11-06’* might be represented internally, as seen in Figure 8-2.

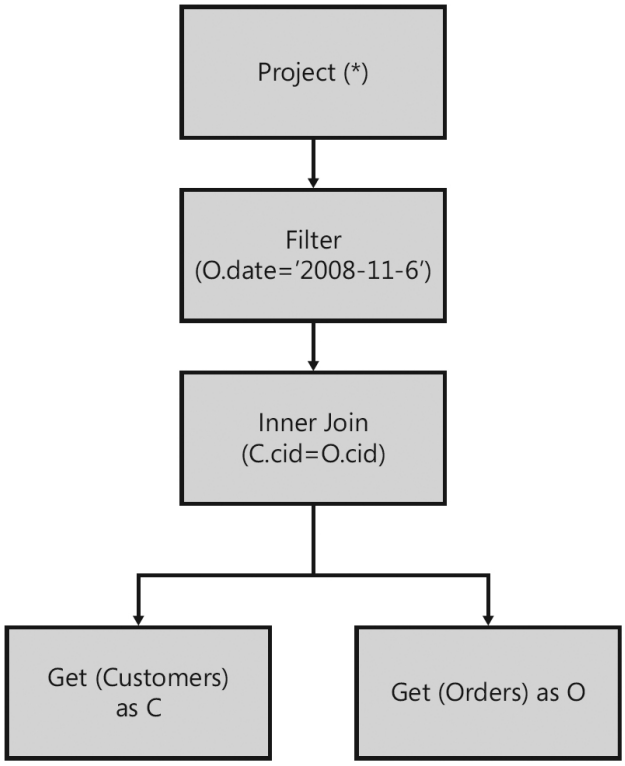


Figure 8-2: Query tree format example

The query processor actually uses different tree formats throughout the compilation process. For example, one job that the Query Optimizer performs is to convert a tree from a logical description of the desired result, as seen previously, to a plan with real physical operators that can be executed. Perhaps the most obvious place where this selection happens is when the Query Optimizer selects a join algorithm, converting a logical join (for example, *INNER JOIN*) into a physical join (a hash join, merge join, or nested loops join). Most of the tree formats are pretty close to each other. In many of the examples in this chapter, optimizer output trees are used to describe specific optimizations performed by the Query Optimizer in earlier, internal tree formats.

What Is Optimization?

So far, we have discussed only the basic transformation from a logical query tree into an equivalent physical query plan. Another major job of the Query Optimizer is to find an *efficient* query plan. There are usually many ways to evaluate a given query, and often some plans are much slower than others. The speed difference between these two plans is so significant that selecting the wrong query plan can cause a database application to perform so slowly that it appears broken to the user. Therefore, it is very important that the Query Optimizer select an efficient plan.

At first, it might seem that there would be an "obvious" best plan for every SQL query, and the Query Optimizer should just select it as quickly as it can. Unfortunately, query optimization is actually a much more difficult problem. Consider the following SQL query:

```
SELECT * FROM A
INNER JOIN B ON (A.a = B.b)
INNER JOIN C ON (A.a = C.c)
INNER JOIN D ON (A.a = D.d)
INNER JOIN E ON (A.a = E.e)
INNER JOIN F ON (A.a = F.f)
INNER JOIN G ON (A.a = G.g)
INNER JOIN H ON (A.a = H.h)
```

This query has many possible implementation plans because inner joins can be computed in different orders. Actually, if you add more tables into this query using this same pattern, the query would have so many possible plan choices that it isn't feasible to consider them all. Because inner joins can be evaluated in any order (ABCD…, ABDC…, ACBD…, …) and in different topologies [(A join B) join (B join C)], the number of possible query plans for this query is actually greater than *N*! [*N* x (*N*-1) x (*N*-2) x …]. As the number of tables in a query increases, the set of alternatives to consider quickly grows to be larger than what can be computed on any computer. The storage of all the possible query plans also becomes a problem. In 32-bit Intel x86-based machines, SQL Server usually has about 1.6 GB of memory that could be used to

compile a query, and it may not be possible to store every possible alternative in memory. Even if a computer could store all these alternatives, the user may not want to wait that long to enumerate all those possible choices. The Query Optimizer solves this problem using heuristics and statistics to guide those heuristics, and this chapter describes these concepts.

Many people believe that it is the job of the Query Optimizer to select the absolute best query plan for a given query. You can now see that the scope of the problem makes this impossible—if you can't consider every plan shape, it is difficult to prove that a plan is optimal. However, it *is* possible for the Query Optimizer to find a "good enough" plan quickly, and often this is the optimally performing plan or very close to it.

How the Query Optimizer Explores Query Plans

The Query Optimizer uses a framework to search and compare many different possible plan alternatives efficiently. This framework allows SQL Server to consider complex, non-obvious ways to implement a given query. Keeping track of all these different alternatives to find a plan to run efficiently is not easy. The search framework of SQL Server contains several components that help it perform its job efficiently and reliably. Although largely internal, these components are described in this section to give you a better idea about how a query is optimized and to better design your applications to take advantage of its capabilities.

Rules

The Query Optimizer is a search framework. From a given query tree, the Query Optimizer considers transformations of that tree from the current state to a different, equivalent state that is also stored in memory. In the framework used in SQL Server, the transformations are done via *rules*. These rules are very similar to the mathematical theorems you likely learned in school. For example, we know that *A INNER JOIN B* is equivalent to *B INNER JOIN A* because both queries return the same result for all possible table data sets. This is a form of commutativity (which, in regular integer arithmetic, means that (1+2) is equivalent to (2+1), meaning that this operation can be performed in any order and yield the same result (or, in the case of databases, return the same set of rows). Rules are matched to tree patterns and are then applied if they are suitable to generate new alternatives (which then may also lead to more rule matching). These rules form the basis of how the Query Optimizer works, and they also help encode some of the heuristics necessary to perform the search in a reasonable amount of time.

The Query Optimizer has different kinds of rules. Rules that heuristically rewrite a query tree into a new shape are called *substitution rules*. Rules that consider mathematical equivalences are called *exploration rules*. These rules generate new tree shapes but cannot be directly executed. Rules that convert logical trees into physical trees to be executed are called *implementation rules.* The best of these generated physical alternatives is eventually output by the Query Optimizer as the final query execution plan.

More Info This chapter contains many examples of query execution plans, used to illustrate the Query Optimizer's behavior. If you would like more background information on how to interpret query execution plans, and what the various operators mean, you can refer to Chapter 3 in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization* (Microsoft Press, 2007). Other than some minor visual changes in the way graphical query plans are displayed in SQL Server 2008, almost all the content in that chapter is applicable to SQL Server 2008. We have made this chapter available for you on the companion Web site (*http://www.SQLServerInternals.com/ companion*).

Properties

The search framework collects information about the query tree in a format that can make it easier for rules to work. These are called *properties,* and they collect information from sub-trees to help make decisions about what rules can be processed at a higher point in a tree. For example, one property used in SQL Server is the set of columns that make up a unique key on the data. Consider the following query:

```
SELECT col1, col2, MAX(col3) FROM Table1 GROUP BY col1, col2;
```

This query is represented internally as a tree, as seen in Figure 8-3.

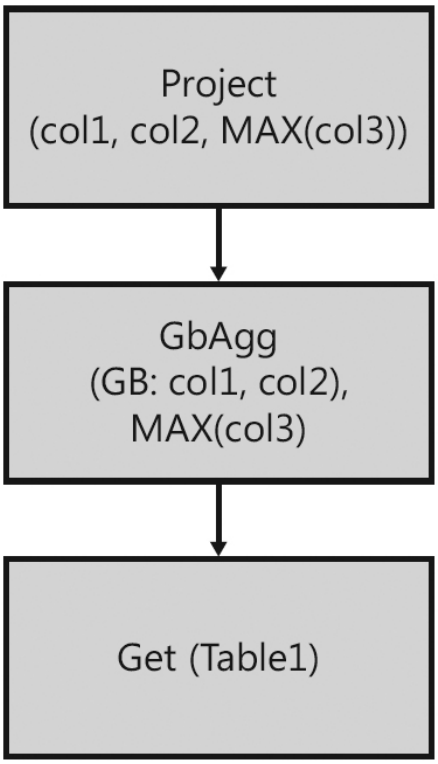


Figure 8-3: GROUP BY tree example

If the columns (*col1*, *col2*) make up a unique key on table *groupby*, then it is not necessary to do grouping at all—each group has exactly one row. The *MAX()* of a set of size one is the element itself. So, it is possible to write a rule that removes the *groupby* from the query tree completely. You can see this rule in action in Figure 8-4.

```
CREATE TABLE groupby (col1 int, col2 int, col3 int);
ALTER TABLE groupby ADD CONSTRAINT unique1 UNIQUE(col1, col2);
SELECT col1, col2, MAX(col3) FROM groupby GROUP BY col1, col2;
```

Figure 8-4: Query plan with aggregate operation removed

If you look at the final query plan, you can see that the Query Optimizer performs no grouping operation even though the query has a *GROUP BY* in it. The properties collected during optimization enable this rule to perform a tree transformation to make the resulting query plan complete more quickly.

SQL Server also collects many properties during optimization. As is done in most modern compilers, the Query Optimizer collects domain constraint information about each column referenced in the query. The Query Optimizer collects information from predicates, join conditions, partitioning information, and check constraints to reason about how all these predicates can be used to optimize the query. One useful application of this scalar property is in *contradiction detection*. The Query Optimizer can determine if the query is written in such a way as to never return any rows at all. When the Query Optimizer detects a contradiction, it actually rewrites the query to remove the portion of the query containing the contradiction. Figure 8-5 contains an example of a contradiction detected during optimization.

```
CREATE TABLE DomainTable(col1 int);
GO
SELECT *
FROM DomainTable D1
INNER JOIN DomainTable D2
ON D1.col1=D2.col1
WHERE D1.col1 > 5 AND D2.col1 < 0;
```

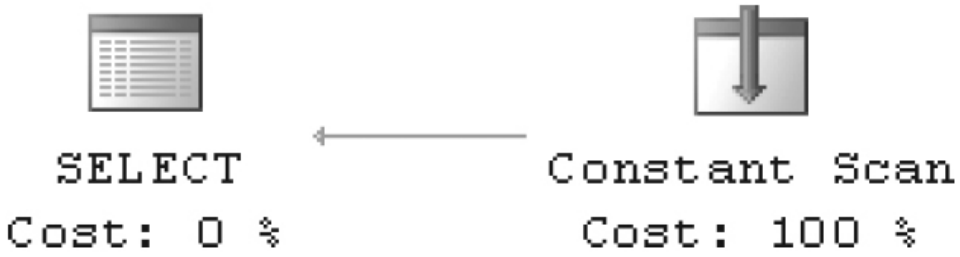


Figure 8-5: Query plan simplified via contradiction

The final query plan does not actually even reference the table at all—it is replaced with a special Constant Scan operator that does not access the storage engine and, in this case, returns zero rows. This means that the query runs faster, takes less memory, and does not need to acquire locks against the resources referenced in the section containing the contradiction when being executed.

> **Note** In this chapter, I have tried to create examples that you can run so you can see for yourself how the system operates based on experiments. Unfortunately, in some cases, different features interact in a way that makes it difficult for me to show you how exactly one feature operates in isolation. In this example, I added a join to avoid another optimization, called *trivial plan,* that sometimes overrides contradiction detection. Because features change from release to release, I ask that you use these examples only to explore the current state of the Query Optimizer—there are no guarantees about how the internals of the Query Optimizer work from release to release, so you should not attempt to build detailed knowledge of the Query Optimizer into your application.

Like rules, there are both logical and physical properties. Logical properties cover things like the output column set, key columns, and whether a column can output any nulls or not. These apply to all equivalent logical and all physical plan fragments. When an exploration rule is evaluated, the resulting query tree shares the same logical properties as the original tree used by the rule. Physical properties are specific to a single plan, and each plan operator has a set of physical properties associated with it. One common physical property is whether the result is sorted. This property would influence whether the Query Optimizer looks for an index to deliver that desired sort. Another physical property is the set of columns from a table that a query can read. This drives decisions such as whether a secondary index is sufficient to return all the needed columns in a query or whether each matching row also needs a base table lookup as well.

Storage of Alternatives—The "Memo"

Earlier in this chapter, I mentioned that the storage of all the alternatives considered during optimization could be large for some queries. The Query Optimizer contains a mechanism to avoid storing duplicate information, thus saving memory (and time) during the compilation process. The structure is called the *Memo*, and one of its purposes is to find previously explored sub-trees and avoid reoptimizing those areas of the plan. It lives for the life of one optimization.

The Memo works by storing equivalent trees in *groups.* If you were to execute each sub-tree in a group, every alternative in that sub-tree would return the same logical result. Conceptually, each operator from the original query tree starts in its own group, meaning that groups reference other groups instead of referencing other operators directly while stored in the Memo. This model is used to avoid storing trees more than once during query optimization, and it enables the Query Optimizer to avoid searching the same possible plan alternatives more than once as well.

In addition to storing equivalent alternatives, groups also store properties structures. Alternatives that are rooted in the same group have equivalent logical and scalar properties. Logical properties are actually called *group properties* in SQL Server, even when not being stored in the Memo. So, every alternative in a group should all have the same output columns, key columns, possible partitionings, and so on. Computing these properties is expensive, so this structure also helps to avoid unnecessary work during optimization.

All considered plans are stored in the Memo. For large queries, the Memo may contain many thousands of groups and many alternatives within each group. Combined, this represents a huge number of alternatives. Although most queries do not consume large amounts of memory during optimization, it is possible that large data warehouse queries could consume all memory on a machine during optimization. If the Query Optimizer is about to run out of memory while searching the set of plans, it contains logic to pick a "good enough" query plan instead of running out of memory.

When the Query Optimizer has finished searching for a plan, it goes through the Memo, starting at the root, to select the best alternative from each group that satisfies the requirements for the query. These operators are assembled into the final

query plan and are then transformed into a format that can be understood by the query execution component in SQL Server. This final tree transformation does contain a small number of run-time optimization rewrites, but it is very close to the showplan output generated for the query plan.

An example of how the Memo works is shown during the examination of the Query Optimizer's architecture and pipeline, later in this chapter.

Operators

SQL Server 2008 has around 40 logical operators and even more physical operators. Some operators are extremely common, such as Join or Filter. Others are harder to find, such as Segment, Sequence Project, and UDX. Operators in SQL Server 2008 follow the model seen in Figure 8-6.

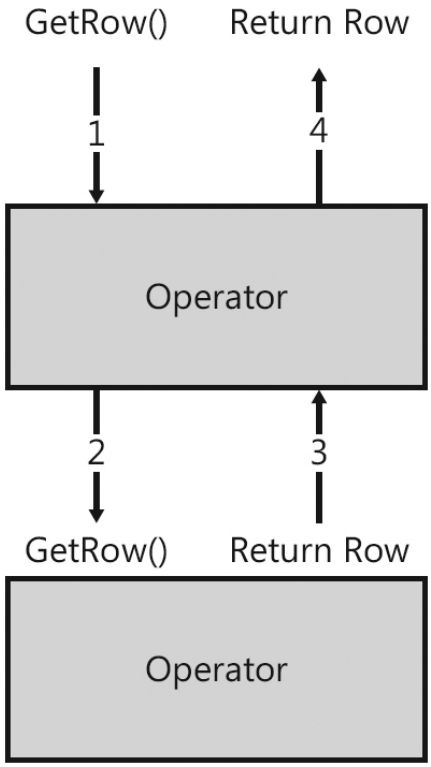


Figure 8-6: SQL Server operator data flow model

Every operator in SQL Server works by requesting rows from one or more children and then producing rows to return to the caller. The caller can be another operator or can be sent to the user if it is the uppermost operator in the query tree. Each operator returns one row at a time, meaning that the caller must call for each row. The uniformity in this design allows operators to be combined in many different ways. It allows new operators to be added to the system without major changes to the Query Optimizer, such as the property framework, that help the Query Optimizer select a query plan.

To make sure that everyone gets the most of this chapter, I'll cover a few of the more rare and exotic operators so that I can reference them later in the chapter, as well as to give you an idea how your query is represented in the system.

Compute Scalar—Project

The Compute Scalar, called a *Project* in the Query Optimizer, is a simple operator that attempts to declare a set of columns, compute some value, or perhaps restrict columns from other operators in the query tree. These correspond to the *SELECT* list in the SQL language. These are actually not overly interesting operations to the Query Optimizer—there is not much that the Query Optimizer needs to do with them. The Query Optimizer ends up moving them around the query tree during optimization, trying to separate them from the rest of the Query Optimizer logic that deals with join order, index selection, and other optimizations.

Compute Sequence—Sequence Project

Compute Sequence is known as a *Sequence Project* in the Query Optimizer, and this operator is somewhat similar to a

Compute Scalar in that it computes a new value to be added into the output stream. The key difference is that this works on an ordered stream and contains state that is preserved from row to row. Ranking functions use this operator, for example. This is implemented using a different physical operation and it imposes additional restrictions on how the Query Optimizer can reorder this expression. This operator is usually seen in the ranking and windowing functions.

Semi-Join

The term *semi-join* comes from the academic database literature, and it is used to describe an operator that performs a join but returns only values from one of its inputs. The query processor uses this internal mechanism to handle most subqueries. SQL Server represents subqueries in this manner because it makes it easier to reason about the set of possible transformations for the query and because the run-time implementation of a semi-join and a regular join are similar. Contrary to popular belief, a subquery is not always executed and cached in a temporary table. It is treated much like a regular join. In fact, the Query Optimizer has transformation rules that can transform regular joins to semi-joins.

One common misconception is that it is inherently incorrect to use subqueries. Like most generalizations, this is not true. Often a subquery is the most natural way to represent what you want in SQL, and that is why it is part of the SQL language. Sometimes, a subquery is blamed for a poorly indexed table, missing statistics, or a predicate that is written in a way that is too obtuse for the Query Optimizer to reason about using its domain constraint property framework. Like everything in life, it is possible to have too many subqueries in a system, especially if they are duplicated many times in the same query. So, if your company's development practices say, "No subqueries," then examine your system a little closely—these are blamed for many other problems that might lie right under the surface.

Listing 8-1 is an example of where a subquery would be appropriate. Let's say that we need to ask a sales tracking system for a store to show me each customer who has made an order in the last 30 days so that we can send them a thank-you e-mail. Figures 8-7, 8-8, and 8-9 show the query plans for the three different approaches to try to submit queries to answer this question.

Listing 8-1: Common Errors in Writing Subquery Plans

```
CREATE TABLE Customers(custid int IDENTITY, name NVARCHAR(100));
CREATE TABLE Orders (orderid INT IDENTITY, custid INT, orderdate DATE, amount MONEY);
INSERT INTO Customers(name) VALUES ('Conor Cunningham');
INSERT INTO Customers(name) VALUES ('Paul Randal');
INSERT INTO Orders(custid, orderdate, amount) VALUES (1, '2008-08-12', 49.23);
INSERT INTO Orders(custid, orderdate, amount) VALUES (1, '2008-08-14', 65.00);
INSERT INTO Orders(custid, orderdate, amount) VALUES (2, '2008-08-12', 123.44);
-- Let's find out customers who have ordered something in the last month

-- Semantically wrong way to ask the question - returns duplicate names (See Figure 8-7)
SELECT name FROM Customers C INNER JOIN Orders O ON C.custid = O.custid WHERE
DATEDIFF("m", O.orderdate, '2008-08-30') < 1

-- and then people try to "fix" by adding a distinct (See Figure 8-8)
SELECT DISTINCT name
FROM
Customers C
INNER JOIN
Orders O
ON C.custid = O.custid
WHERE DATEDIFF("m", O.orderdate, '2008-08-30') < 1;
-- this happens to work, but it is fragile, hard to modify, and it is usually not done
properly.

-- the subquery way to write the query returns one row for each matching Customer
SELECT name
FROM Customers C
WHERE
EXISTS (
SELECT 1
FROM Orders O
WHERE C.custid = O.custid AND DATEDIFF("m", O.orderdate, '2008-08-30') < 1
);
-- note that the subquery plan has a cheaper estimated cost result
-- and should be faster to run on larger systems
```

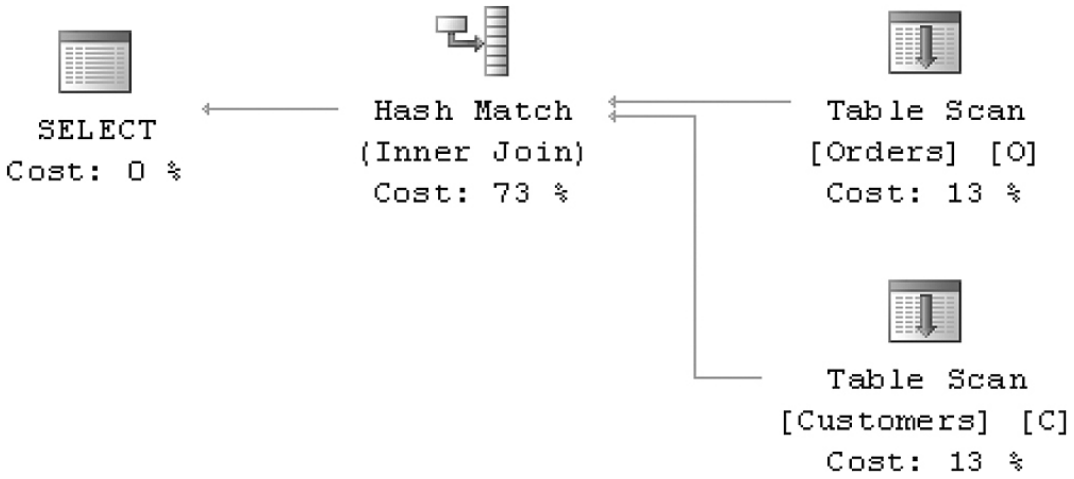


Figure 8-7: Plan for query using INNER JOIN instead of subquery

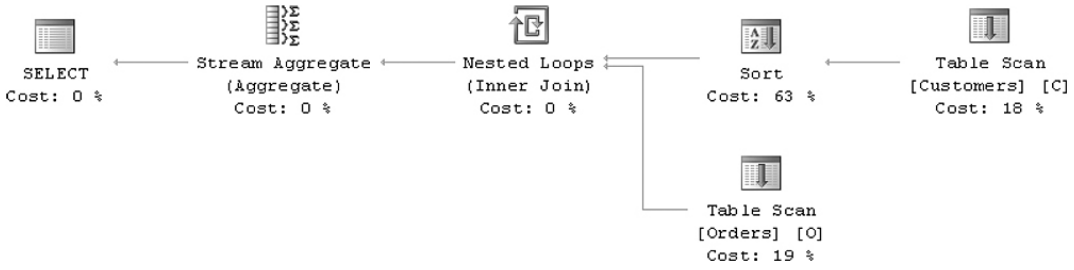


Figure 8-8: Plan for query using DISTINCT and INNER JOIN instead of subquery

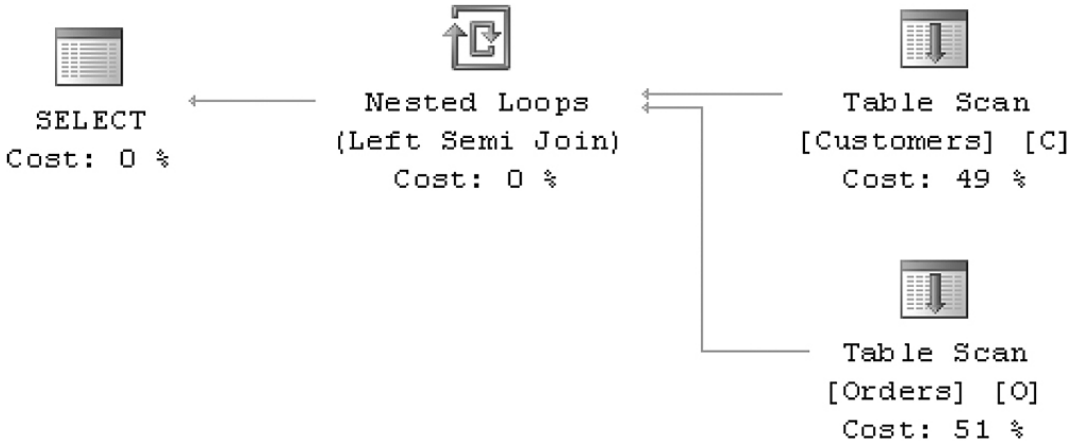


Figure 8-9: Plan for query using subquery

In this last query plan, the matching rows from the *Customers* table are kept and directly returned to the user through the Left Semi-Join operator.

> **Note** The Left and Right Semi-Join have to do with which child's rows are preserved in the operation. Unfortunately for anyone confused as to the meaning of these operators, the plan representation in SQL Server Management Studio and in previous tools is transposed. The "left" child is the top child and the "right" child is the bottom child in the transposed form.

Apply

CROSS APPLY and *OUTER APPLY* were added to SQL Server 2005, and they represent a special kind of subquery where a value from the left input is passed as a parameter to the right child. This is sometimes called a *correlated nested loops join,* and it represents passing a parameter to a subquery. The most common application for this feature is to do an index lookup join, as seen in Listing 8-2 and Figure 8-10.

Listing 8-2: Example of APPLY Query

```
CREATE TABLE idx1(col1 INT PRIMARY KEY, col2 INT);
CREATE TABLE idx2(col1 INT PRIMARY KEY, col2 INT);
GO
SELECT *
FROM idx1
CROSS APPLY (
    SELECT *
    FROM idx2
    WHERE idx1.col1=idx2.col1
  ) AS a;
```

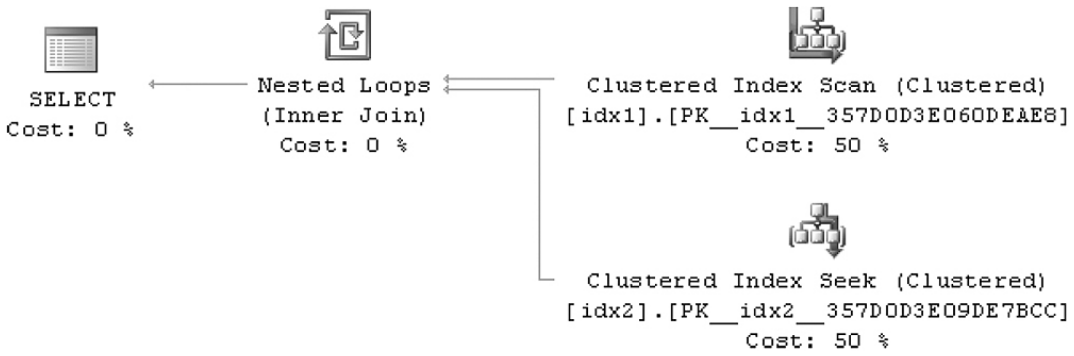


Figure 8-10: APPLY query plan

This query is logically equivalent to an *INNER JOIN*, and Figure 8-11 demonstrates that the resulting query plan is identical in SQL Server 2008.

```
SELECT * FROM idx1 INNER JOIN idx2
ON idx1.col1=idx2.col1;
```

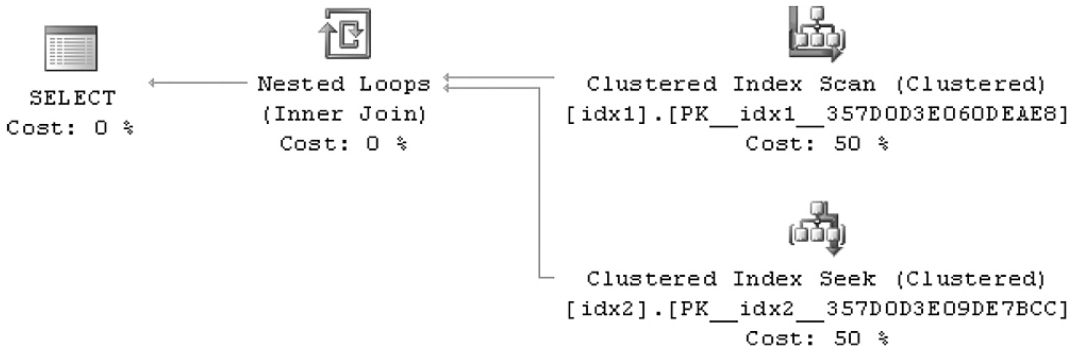


Figure 8-11: INNER JOIN query generates a nested loop and seek plan

In both cases, a value from the outer table is referenced as an argument to the seek on the inner table. Notice that a standard inner join is also able to generate a seek, which means that the Query Optimizer considers turning a JOIN into an APPLY as part of the optimization process. Although the example I have done here is so simple that you would not need to write the query in the way that I did, there are more complex scenarios where this syntax is useful. First, there is a common pattern for dynamic management views (DMVs, including an example in the section entitled "Plan Hinting," later in this chapter) where a value is passed to a management function using a cross apply. Second, there may be rare, very complex cases when the Query Optimizer's rule engine cannot rewrite a simple inner join to get an index seek. In those cases, rewriting the query to use *CROSS APPLY* is useful to pass a parameter down past an opaque operator manually. The semantics of a query can change as a result of a rewrite like this, so be very sure that you understand the semantics of your query when considering a rewrite like this.

The Apply operator is almost like a function call in a procedural language. For each row from the outer (left) side, some logic on the inner (right) side is evaluated and zero or more rows are returned for that invocation of the right sub-tree. The Query Optimizer can sometimes remove the correlation and convert an Apply into a more general join, and in those cases other joins can sometimes be reordered to explore different plan choices.

Spools

SQL Server has a number of different, specialized spools. Each one is highly tuned for some scenario. Conceptually, they all do the same thing—they read all the rows from the input, store it in memory or spill it to disk, and then allow operators to read the rows from this cache. Spools exist to make a copy of the rows, and this can be important for transactional consistency in some update plans and to improve performance by caching a complex subexpression to be used multiple times in a query.

The most exotic spool operation is called a *common subexpression spool*. This spool has the ability to be written once and then read by multiple, different children in the query. This is currently the only operator that can have multiple parents in the final query plan. This spool shows up multiple times in the showplan output and it is actually the *same* operator. Common subexpression spools have only one client at a time. So, the first instance populates the spool, and each later reference reads from this spool in sequence. The first reference has children, while later references appear in the query plan as leaves of the query tree.

Common subexpression spools are used most frequently in wide update plans, described later in this chapter. However, they are also used in windowed aggregate functions. These are special aggregates that do not have to collapse the rows like a regular aggregate computation. Listing 8-3 and Figure 8-12 demonstrate how a common subexpression spool is used to store the intermediate query input and then use it multiple times as inputs to other parts of the query tree. The initial table spool reads values from *window1*, and the later branches in the tree supply the spooled rows to multiple branches.

Listing 8-3: Aggregate with OVER Clause Uses Common Subexpression Spool

```
CREATE TABLE window1(col1 INT, col2 INT);
GO
DECLARE @i INT=0;
WHILE @i<100
BEGIN
INSERT INTO window1(col1, col2) VALUES (@i/10, rand()*1000);
SET @i+=1;
END;

SELECT col1, SUM(col2) OVER(PARTITION BY col1) FROM window1;
```

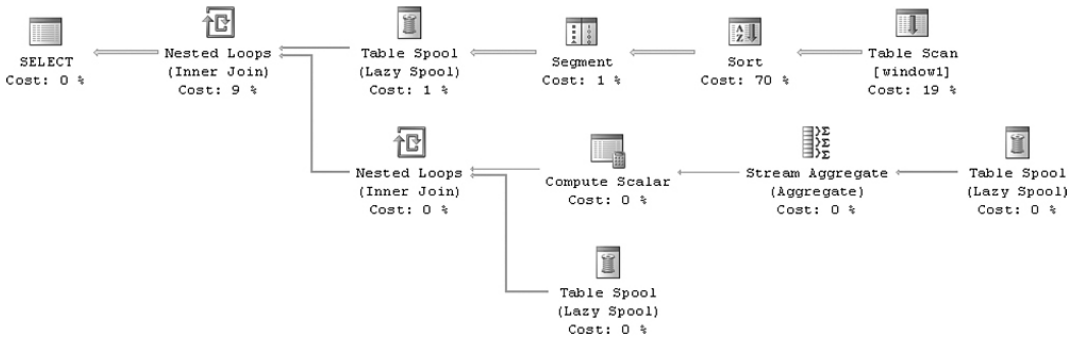


Figure 8-12: Query plan containing common subexpression spool

Exchange

The Exchange operator is used to represent parallelism in query plans. This can be seen in the showplan as a Gather Streams, Repartition Streams, or Distribute Streams operation, based on whether it is collecting rows from threads or distributing rows to threads, respectively. Several row distribution algorithms exist, and each operator has a preferred algorithm based on its context in a query. In SQL Server, parallelism exists in zones where the system tries to speed up by using additional CPUs. Figure 8-13 demonstrates a query where multiple threads scan a table in parallel.

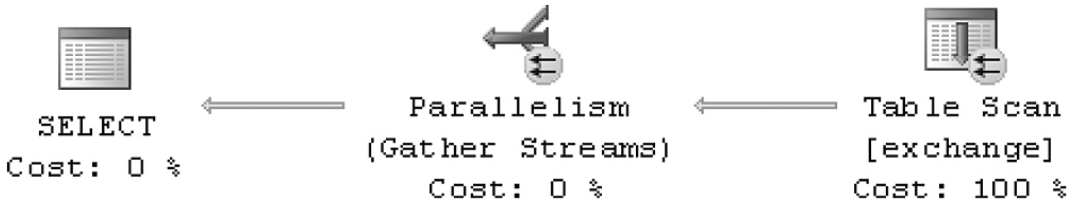


Figure 8-13: Exchange operator in query plan

More Info Other SQL Server 2008 operators are described online at *http://technet.microsoft.com/en-us/library/ms191158. aspx*.

Optimizer Architecture

The Query Optimizer contains many optimization phases that each performs different functions. The different phases help the Query Optimizer perform the highest-value operations earliest in the optimization process.

The major phases in the optimization of a query, as shown in Figure 8-14, are as follows:

- Simplification
- Trivial plan
- Auto-stats create/update
- Exploration/implementation (phases)
- Convert to executable plan

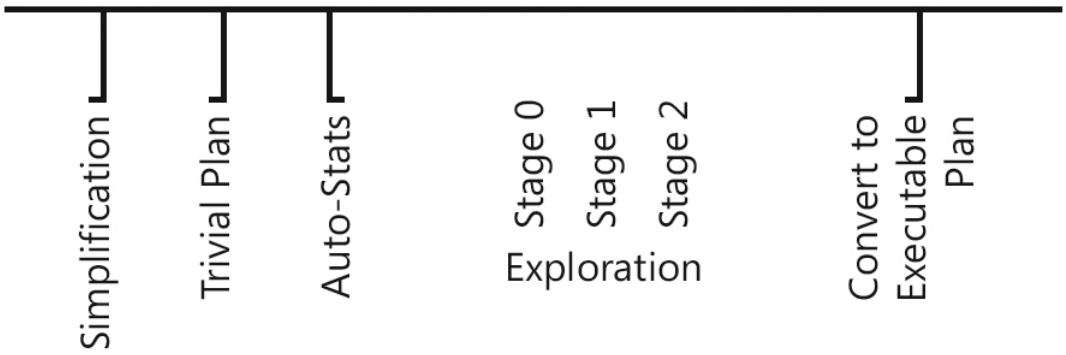


Figure 8-14: Query Optimizer pipeline

Before Optimization

The SQL Server query processor performs several steps before the actual optimization process begins. These transformations help shape the tree into a form about which it can be easily reasoned. View expansion is one major preoptimization activity. When a query is compiled that references a view, the text of the view is read from the server's metadata and parsed as well. One consequence of this design choice is that a query that references a view many times gets this view expanded many times before it is optimized. Coalescing adjacent *UNION* operations is another preoptimization transformation that is performed to simplify the tree. This converts the syntactic two-child form of *UNION [ALL], INTERSECT [ALL],* and *EXCEPT [ALL]* into a single operator that can have more than two children. This rewrite simplifies the tree structure and makes it easier for the Query Optimizer to write rules to affect *UNION*s. For example, grouping *UNION* operations makes the task of removing duplicate rows easier and more efficient.

Simplification

Early in optimization, the tree is normalized in the Simplification phase to convert the tree from a form linked closely to the user syntax into one that helps later processing. For example, the Query Optimizer detects semantic contradictions in the query and removes them by rewriting the query into a simpler form. In addition, the rewrites performed in this section make subsequent operations such as index matching, computed column matching, and statistics generation easier to perform correctly.

The Simplification phase performs a number of other tree rewrites as well. These activities include

- Grouping joins together and picking an initial join order, based on cardinality data for each table
- Finding contradictions in queries that can allow portions of a query not to be executed
- Performing the necessary work to rewrite *SELECT* lists to match computed columns.

A contradiction detection example was shown earlier in this chapter in Figure 8-5.

Trivial Plan/Auto-Parameterization

The main optimization path in SQL Server is a very powerful cost-based model of the execution time of a query. As databases and queries over those databases have become larger and more complex, this model has allowed SQL Server to solve bigger and bigger business problems. The fixed startup cost for running this model can be expensive for applications that are not trying to perform complex operations. Making a single path that spans from the smallest to the largest queries can be challenging, as the requirements and specifications are vastly different.

To be able to satisfy small query applications well, a fast path was added to SQL Server to identify queries where cost-based optimization was not needed. Generally, this code identifies cases where a query does not have any cost-based choices to make. This means that there is only one plan to execute or there is an obvious best plan that can be identified. In these cases, the Query Optimizer directly generates the best plan and returns it to the system to be executed. For example, the query *SELECT col1 FROM Table1* for a table without any indexes has a straightforward best plan choice—read the rows from the base table heap and return them to the user, as seen in Figure 8-15.

```
CREATE TABLE Table1 (col1 INT, col2 INT);
SELECT col1 FROM Table1;
```

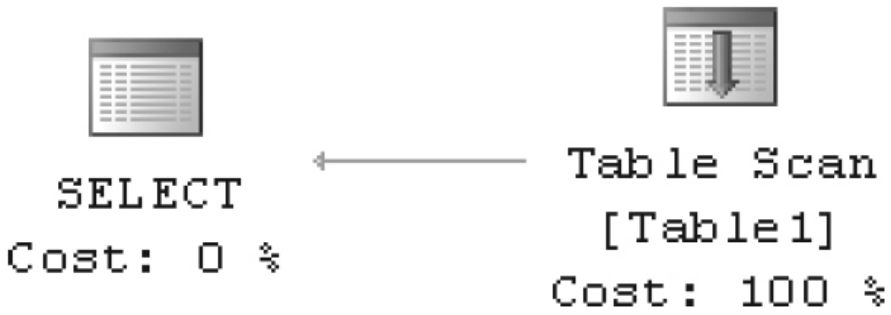


Figure 8-15: Trivial plan example—table scan

The SQL Server query processor actually takes this concept one step further. When simple queries are compiled and optimized, the query processor attempts to turn these queries into a parameterized query. If the plan is determined to be trivial, the parameterized query is turned into an executable plan. Then, future queries that have the same shape except for constants in well-known locations in the query just run the existing compiled query and avoid going through the Query Optimizer at all. This speeds up applications with small queries on SQL Server significantly.

```
SELECT col1 FROM Table1 WHERE col2 = 5;
SELECT col1 FROM Table1 WHERE col2 = 6;
```

If you look at the text of these queries in the procedure cache in Listing 8-4, you see that there is actually only one query plan, and it is parameterized.

Listing 8-4: Automatically Parameterized Query Text

```
SELECT text
    FROM sys.dm_exec_query_stats AS qs
    CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
WHERE st.text LIKE '%Table1%';

---------------------------------------
(@1 tinyint)SELECT [col1] FROM [Table1] WHERE [col2]=@1
```

If you examine the XML plan for this query plan, you see that there is an indication that this query was a trivial plan. The other choice is full, meaning that cost-based optimization was performed):

```
. . . <StmtSimple . . . StatementOptmLevel="TRIVIAL"> . . .
```

(XML plan output is verbose, so I have omitted most of it for space. The bold code shows the choice the Query Optimizer made.)

Limitations

The trivial plan optimization was introduced in SQL Server 7.0. While each version of SQL Server has slightly different rules, all versions have queries that skip the trivial plan stage completely and only perform regular optimization activities. Using more complex features can disqualify a query from being considered trivial because they always have a cost-based plan choice or are too difficult to identify as trivial or not. Examples of query features that cause a query not to be considered trivial include Distributed Query, Bulk Insert, XPath queries, queries with joins or subqueries, and queries with hints, some cursor queries, and queries over tables containing filtered indexes.

SQL Server 2005 added another feature, called *forced parameterization*, to auto-parameterize queries more aggressively. This feature parameterizes all constants, ignoring cost-based considerations. This feature is most useful for an application where the SQL is generated (and you cannot make it generate parameterized queries) and the resulting query plans are almost always identical (or the plans perform similarly even if they differ). Specifically, it is worth considering when the application queries cannot be changed by the DBA in charge of the server.

The benefit of this feature is that it can reduce compilations, compilation time, and the number of plans in the procedure cache. All these things *can* improve system performance. On the other hand, this feature can reduce performance when different parameter values would cause different plans to be selected. These values are used in the Query Optimizer's cardinality and property framework to make decisions about how many rows will be returned from each possible plan choice, and forced parameterization blocks these optimizations. So if you think that your application would benefit from using forced parameterization, perform some experiments and see whether the application works better. Chapter 9, "Plan Caching and Recompliation," goes into more detail on the various parameterization options.

The Memo—Exploring Multiple Plans Efficiently

The core structure of the Query Optimizer is the Memo. This structure helps store the result of all the rules that are run in the Query Optimizer, and it also helps guide the search of possible plans to find a good plan quickly and avoid searching a sub-tree more than once. This speeds up the compilation process and reduces the memory requirements. In effect, this allows the Query Optimizer to run more advanced optimizations compared to other optimizers without a similar mechanism. Although this structure is internal to the Query Optimizer, this section describes its basic operations so that you can better understand the way that the Query Optimizer selects a plan.

The Memo stores operators from a query tree and uses logical pointers to represent the edges of that tree. If we consider the query *SELECT * FROM (A INNER JOIN B ON A.a=B.b) AS D INNER JOIN C ON D.c=C.c,* this can be drawn as a tree, as seen in Figure 8-16.

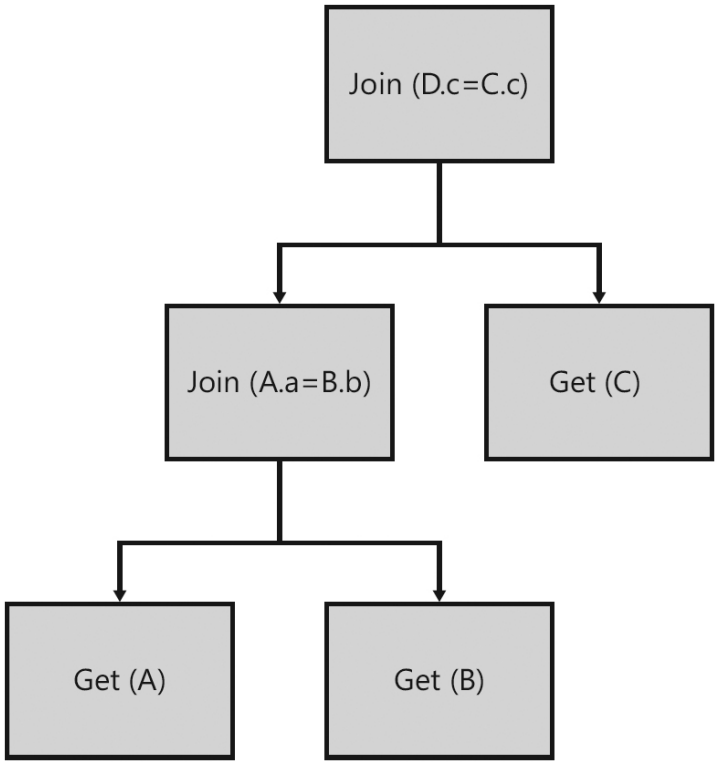


Figure 8-16: Tree of a three-table join

The same query stored in the Memo can be seen in Figure 8-17.

```
(Root) Group 4:
          0 Join 3 2
Group 3:
          0 Join 0 1
Group 2:
          0 Table C
Group 1:
          0 Table B
Group 0:
          0 Table A
```

Figure 8-17: Initial Memo layout for a three-table join

The Memo is made up a series of *groups*. When the Memo is first populated, each operator is put into its own group. The references between operators are changed to be references to other groups in the Memo. In this model, it is possible to store multiple alternatives that yield the same result in the same group in the Memo. With this change, it is possible to search for the best sub-tree independently of what exists in higher-level groups in the Memo. Logical properties are stored within each memo group, and every additional entry in a group can share the property structure for that group with the initial alternative.

One type of alternative explored by the Query Optimizer is *join associativity*. [(A join B) join C)] is equivalent to [A join (B join C)]. After this transformation is considered by the Query Optimizer, Figure 8-18 describes the updated Memo structure. (The bold sections are new.)

```
Group 5:
          0 Join 1 2
(Root) Group 4:
          1 Join 0 5
          0 Join 3 2
```

```
Group 3:
          0 Join 0 1
Group 2:
          0 Table C
Group 1:
          0 Table B
Group 0:
          0 Table A
```

Figure 8-18: Three-table Memo after the join associativity rule has been applied

Notice how the new alternative fits into a structure. (B join C) that has not been previously seen in the Memo, so a new group is created that then references the existing groups for B and C. This representation saves a lot of memory when considering multiple possible query plans, and it makes it possible for the Query Optimizer to know if it has previously considered a section of the search space so that it can avoid doing that work again. (A join C) would be another valid alternative, though it is not shown.

Rules are the mechanisms that allow the Memo to explore new alternatives during the optimization process. The join associativity example is implemented as an optimization rule that matches a specific pattern and then creates a new alternative that is equivalent to the first one (returns the same result for that portion of the query). The result of a rule, by definition, can go into the same group as the root of the original pattern.

An optimization search pass is split into two parts. In the first part of the search, exploration rules match logical trees and generate new, equivalent alternative logical trees that are inserted into the Memo. Implementation rules run next, generating physical trees from the logical trees. Once a physical tree has been generated, it is evaluated by the costing component to determine the cost for this query tree. The resulting cost is stored in the Memo for that alternative. When all physical alternatives and their costs are generated for all groups in the Memo, the Query Optimizer finds the one query tree in the Memo that has the lowest cost and copies that into a stand-alone tree. The selected physical tree is very close to the showplan form of the tree.

The optimization process is optimized further by using multiple search passes, separating the rules based on cost and how likely they are to be useful. There are three phases, and each phase runs a set of exploration and implementation rules. The phases are configured to make small queries optimize quickly and to make more expensive queries consider more aggressive rewrite rules that may take longer to compile. For example, index matching is performed in the first phase, whereas the matching of index view is generally not performed until a later stage. The Query Optimizer can quit optimization at the end of a phase if a sufficiently good plan has been found. This calculation is done by comparing the estimated cost of the best plan found so far against the actual time spent optimizing so far. If the current best plan is still very expensive, then another phase is run to try to find a better plan. This model allows the Query Optimizer to generate plans efficiently for a wide range of workloads.

Convert to Executable Plan

At the end of the search, the Query Optimizer has selected a single plan to be returned to the system. This plan is copied from the Memo into a separate tree format that can be stored in the Procedure Cache. During this process, a few small, physical rewrites are performed. Finally, the plan is copied into a new piece of contiguous memory and is stored in the procedure cache.

Statistics, Cardinality Estimation, and Costing

The Query Optimizer uses a model with estimated costs of each operator to determine which plan to choose. The costs are based on statistical information used to estimate the number of rows processed in each operator. By default, statistics are generated automatically during the optimization process to help generate these cardinality estimates. The Query Optimizer also determines which columns need statistics on each table.

Once a set of columns is identified as needing statistics, the Query Optimizer attempts to find a preexisting statistics object for that column. If it doesn't find one, the system samples the table data to create a new statistics object. If one already existed, it is examined to determine if the sample was recent enough to be useful for the compilation of the current query. If it is considered out of date, a new sample is used to rebuild the statistics object. This process continues for each column where statistics are needed.

Both auto-create and auto-update statistics are enabled by default. In practice, most people leave these flags enabled and get good behavior from the Query Optimizer. However, it is possible to disable the creation and update behavior of

statistics:

```
ALTER DATABASE ... SET AUTO_CREATE_STATISTICS {ON | OFF }
ALTER DATABASE ... SET AUTO_UPDATE_STATISTICS {ON | OFF }
```

These commands modify the behavior for auto-create and auto-update statistics, respectively, for all tables in a database. If automatic creation or updating of statistics is disabled, the Query Optimizer returns a warning in the showplan output when compiling a query where it thinks it needs this information. In this mode of operation, it would be the responsibility of the DBA to keep the statistics objects up to date in the system.

It is also possible to control the auto-update behavior of individual statistics objects using hints on specific operations:

```
CREATE INDEX ... WITH (STATISTICS_NORECOMPUTE = ON)
CREATE STATISTICS ... WITH (NORECOMPUTE)
```

While these settings are usually left enabled, some reasons for disabling creating or updating statistics include the following:

- The database has a maintenance window when the DBA has decided to update statistics explicitly instead of having these objects update automatically during the day. This is often because the DBA has reason to believe that the Query Optimizer may choose a poor plan if the statistics are changed.
- The database table is very large, and the time to update the statistics automatically is too high.
- The database table has many unique values, and the sample rate used to generate statistics is not high enough to capture all the statistical information needed to generate a good query plan. The DBA likely uses a maintenance window to update statistics manually at a higher sample rate than the default (which varies based the size of the table).
- The database application has a short query timeout defined and does not want automatic statistics to cause a query to take noticeably longer than average to compile because it could cause that timeout to abort the query.

This last scenario manifests in a subtle manner that can break your applications. If a query in an OLTP application was set with a small timeout of a few seconds, this is generally sufficient to compile all queries (even with automatic statistics). However, as the database table grows, the time to sample the table to create or update statistics grows. Eventually, the total time to perform this operation reaches the query timeout. Because each query is compiled as part of a user transaction, a timeout forces the transaction to abort and roll back. When the next query against that table was compiled, the timeout is hit again and the whole query roll backs. This unfortunately caused applications to fail unexpectedly, and often this would happen after an application was deployed because it was just some timing threshold based on database size.

To address this functionality, SQL Server 2005 introduced a feature called *asynchronous statistics update (ALTER DATABASE … SET AUTO_UPDATE_STATISTICS_ASYNC {ON | OFF)*. This allows the statistics update operation to be performed on a background thread in a different transaction context. The benefit of this model is that it avoids the repeating rollback issue. The original query continues and uses out-of-date statistical information to compile the query and return it to be executed. When the statistics are updated, plans based on those statistics objects are invalidated and are recompiled on their next use.

Statistics Design

Statistics are stored in the system metadata. They are composed primarily of a histogram (a representation of the data distribution for a column). Do not get this confused: sometimes people say *statistics* when they mean *histogram*. Other elements in the statistics object include some header information (including the number of rows sampled when the object was created), trie trees (a representation of the data distribution for string columns), and density information (which tracks information about average data distributions across one or more columns).

Statistics can be created over most, but not all, data types in SQL Server 2008. As a general rule, data types that support comparisons (such as *>, =,* and so on) support the creation of statistics. The Query Optimizer doesn't need to reason about distributions if these are not comparable in the language. Examples of data types where statistics are not supported include old-style BLOBs (such as *image, text,* and *ntext*) and some of the newer user-defined data type (UDT)–based types when they are not byte-order comparable.

In addition, SQL Server supports statistics on computed columns. This allows the Query Optimizer to make cardinality estimates over expressions, such as *col1 + col2*, or some of the more complex types, such as the geography type, where the primary use case is to run a function on the UDT instead of comparing the UDT directly.

Listing 8-5 creates statistics on a persisted computed column created on a function of an otherwise non-comparable UDT. When this UDT method is used in later queries, the Query Optimizer can use this statistic to estimate cardinality more accurately.

Listing 8-5: DBCC SHOW_STATISTICS over a Persisted Computed Column

```
CREATE TABLE geog(col1 INT IDENTITY, col2 GEOGRAPHY);
INSERT INTO geog(col2) VALUES (NULL);
INSERT INTO geog(col2) VALUES (GEOGRAPHY::Parse('LINESTRING(0 0, 0 10, 10 10, 10 0, 0 0)'));
ALTER TABLE geog ADD col3 AS col2.STStartPoint().ToString() PERSISTED;
CREATE STATISTICS s2 ON geog(col3);
DBCC SHOW_STATISTICS('geog', 's2');
```

Statistics can be enumerated by querying the system metadata using the following code, and the results are shown in Figure 8-19.

```
SELECT o.name AS tablename, s.name AS statname
FROM sys.stats s INNER JOIN sys.objects o ON s.object_id = o.object_id;
```

Results | Messages

	tablename	statname
86	Customers	_WA_Sys_00000001_00551192
87	Customers	_WA_Sys_00000002_00551192
88	Orders	_WA_Sys_00000003_014935...
89	Orders	_WA_Sys_00000002_014935...
90	idx1	PK__idx1__357D0D3E060DE...
91	idx2	PK__idx2__357D0D3E09DE7...
92	idx2	i1
93	geog	s2
94	queue_me...	queue_clustered_index
95	queue_me...	queue_secondary_index
96	queue_me...	queue_clustered_index
97	queue_me...	queue_secondary_index
98	queue_me...	queue_clustered_index

Figure 8-19: Query output listing statistics objects

Once identified, the statistics object can be viewed using the *DBCC SHOW_STATISTICS* command, as shown in Figure 8-20.

```
DBCC SHOW_STATISTICS(exchange,_WA_Sys_00000004_4C0144E4)
```

Results | Messages

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	_WA_Sys_00000004_4C0144E4	Nov 27 2008 9:49AM	60000	8264	185	0.703127	4	NO	NULL	60000

	All density	Average Length	Columns
1	0.0001049208	4	col4

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	0	0	1	0	1
2	23	102.4228	28.55033	22	4.65558
3	80	321.9001	20.53245	52	6.17722
4	119	263.3728	20.53245	38	6.930864
5	161	226.7933	20.53245	41	5.531543
6	192	204.8455	20.53245	30	6.828184
7	229	146.3182	20.53245	29	5.028785
8	268	219.4774	20.53245	38	5.77572
9	319	314.5842	20.53245	50	6.291684
10	357	234.1092	28.55033	34	6.80407
11	396	182.8978	20.53245	38	4.819702
12	423	190.2137	28.55033	26	7.315912
13	467	197.5296	28.55033	33	6.052347
14	521	336.5319	20.53245	53	6.349659
15	567	314.5842	20.53245	45	6.99076
16	621	270.6887	20.53245	50	5.377159
17	672	256.0569	35.93245	45	5.68644
18	699	139.0023	20.53245	22	6.31407
19	743	204.8455	28.55033	38	5.398066
20	780	182.8978	11.53645	36	5.080494
21	828	292.6365	20.53245	47	6.226308
22	867	197.5296	20.53245	36	5.46
23	911	219.4774	28.55033	38	5.783642
24	938	131.6864	20.53245	26	5.152974
25	1001	270.6887	20.53245	50	5.377159

Figure 8-20: DBCC SHOW_STATISTICS output

Once the Query Optimizer has determined that it needs to either create a new statistics object or update an existing one that is out of date, the system creates an internal query to generate a new statistics object. Figure 8-21 demonstrates the query that builds the statistics object in the SQL Server Profiler output.

```
Showplan All For Query Compile                                                Micr

StmtText                                                                      StmtId
--------                                                                      ------
Stream Aggregate(DEFINE:([Expr1004]=STATMAN([s1].[dbo].[trace].[col1])))      0
  |--Sort(ORDER BY:([s1].[dbo].[trace].[col1] ASC))                           0
       |--Table Scan(OBJECT:([s1].[dbo].[trace]))                             0
```

Figure 8-21: SQL Profiler showplan output for histogram generation

Note *STATMAN* is a special internal aggregate function that works like other aggregate functions in the system—many rows are consumed by a streaming group by operator and are passed to the *STATMAN* aggregate. It generates a BLOB that stores the histogram, density information, and any trie trees created during this operation. When finished, the statistics blob is stored in the database metadata and is used by queries (including the one that issued the command originally, except in the case of asynchronous statistics update).

The Optimizer samples database pages to generate statistics, including all rows from each sampled page. For small tables, all pages are sampled (meaning all rows are considered when building the histogram). For larger tables, a smaller and smaller percentage of pages are sampled. To keep the histogram a reasonable size, it is limited to 200 total steps. If it examines more than 200 unique values while building the histogram, the Query Optimizer uses logic to try to reduce the number of steps based on an algorithm that preserves as much distribution information as possible. Because histograms are most useful for capturing the non-uniform data distributions of a system, this means that it tries to preserve the information that captures the most frequent values and how much more frequent they are than the least frequent values in the data.

Density/Frequency Information

In addition to a histogram, the Query Optimizer also keeps track of the number of unique values for a set of columns. When combined with the total number of rows viewed when creating the table, this can calculate the average number of duplicate values in the column. This information, called the *density information,* is stored in the histogram. Density is calculated by the formula *1/frequency,* with *frequency* being the average number of duplicates for each value in a table. This information

is also returned when one calls *DBCC SHOW_STATISTICS*. For multicolumn statistics, the *statistics* object stores density information for each combination of columns (in the order that they were specified in the *CREATE STATISTICS* statement) in the *statistics* object. This stores information about the number of duplicate *sets* of values.

In Listing 8-6, we will create a two-column table with 30,000 rows.

Listing 8-6: Multicolumn Statistics

```
CREATE TABLE MULTIDENSITY (col1 INT, col2 INT);
go
DECLARE @i INT;
SET @i=0;
WHILE @i < 10000
BEGIN
       INSERT INTO MULTIDENSITY(col1, col2) VALUES (@i, @i+1);
       INSERT INTO MULTIDENSITY(col1, col2) VALUES (@i, @i+2);
       INSERT INTO MULTIDENSITY(col1, col2) VALUES (@i, @i+3);
       set @i+=1;
END;
GO
-- create multi-column density information
CREATE STATISTICS s1 ON MULTIDENSITY(col1, col2);
GO
```

In *col1*, there are 10,000 unique values, each duplicated three times. In *col2*, there are actually 10,002 unique values. For the multicolumn density, each set of (*col1, col2*) in the table is unique. Figure 8-22 shows the data stored for the multicolumn statistics object.

```
DBCC SHOW_STATISTICS ('MULTIDENSITY', 's1')
```

	Name	Updated	Rows	Rows Sampled	Steps	Density
1	s1	Nov 27 2008 10:16AM	30000	30000	3	0.3333333

	All density	Average Length	Columns
1	0.0001	4	col1
2	3.333333E-05	8	col1, col2

Figure 8-22: Multicolumn density information in the statistics object

The density information for *col1* is 0.0001. 1/0.0001 = 10,000, which is the number of unique values of *col1*. The density information for (*col1, col2*) is about 0.00003 (the numbers are stored as floating point and are imprecise).

Let's examine the cardinality estimates for the *GROUP BY* operation using *GROUP BY* lists that match the density information in Figure 8-23. The actual and estimated cardinalities match up exactly for this query.

```
SET STATISTICS PROFILE ON
SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP BY col1
```

	Rows	Executes	StmtText	EstimateRows
1	10000	1	SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP ...	10000
2	0	0	|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(i...	10000
3	10000	1	|--Hash Match(Aggregate, HASH:([s1].[dbo].[MULTIDENSI...	10000
4	30000	1	|--Table Scan(OBJECT:([s1].[dbo].[MULTIDENSITY]))	30000

Figure 8-23: STATISTICS PROFILE output for the hash aggregate

Note I have reordered the columns from the *STATISTICS PROFILE* output to show the *EstimateRows* column for the Hash Match implementing the *GROUP BY* operation.

For a query grouping over both columns, you can see that the estimate matches up with the value seen in the density

calculation. The *STATISTICS PROFILE* output in Figure 8-24 shows that this changes the estimate to 30,000 rows.

```
SET STATISTICS PROFILE ON
SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP BY col1, col2
```

Rows	Executes	StmtText	EstimateRows
30000	1	SELECT COUNT(*) AS CNT FROM MULTIDENSITY GROUP ...	30000
0	0	|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(i...	30000
30000	1	|--Hash Match(Aggregate, HASH:([s1].[dbo].[MULTIDENSI...	30000
30000	1	|--Table Scan(OBJECT:([s1].[dbo].[MULTIDENSITY]))	30000

Figure 8-24: The STATISTICS PROFILE output for a two-column aggregate

The Query Optimizer actually has to perform an additional step when calculating the output cardinality for an operator. Because statistics are usually created before the compilation of the query that uses them and are often only samples of the data, the values stored in the statistics object do not usually match the exact count of rows at the time the query is compiled. So the Query Optimizer uses these two values to calculate the fraction of rows that should qualify in the operation. This is then scaled to the actual number of values in the table at the time that the query is compiled.

The Query Optimizer does not expose exactly how each part of the cardinality estimate is computed. However, if you find that a query has estimates that vary widely from what actually happens when you run the query, statistics profile can help you identify if the Query Optimizer has bad information. You may need to update statistics to capture new data in the table, create statistics with a higher sample rate, or otherwise make sure that the information used during compilation is accurate. Although SQL Server does this automatically in most cases, this is often a good way to find and fix problems with poor plan selection.

Filtered Statistics

As part of the Filtered Index feature added in SQL Server 2008, the Filtered Statistics feature was also added. This means that the *statistics* object is created over a subset of the rows in a table based on a filter predicate. Creating a filtered index auto-creates a matching filtered *statistics* object that matches the behavior of nonfiltered indexes. This information is exposed through the *sys.stats* metadata view shown in Figure 8-25.

```
SELECT * FROM SYS.STATS
```

	object_id	name	stats_id	auto_created	user_created	no_recompute	has_filter	filter_definition
95	26157...	s1	2	0	1	0	0	NULL
96	26157...	_WA_Sys_00000002_0F975522	3	1	0	0	0	NULL
97	26157...	s3	4	0	1	0	1	([col2]>(5))
98	19930...	queue_clustered_index	1	0	0	0	0	NULL

Figure 8-25: The filter_definition expression in SQL Server 2008 Statistics

Filtered statistics are used in a manner that is similar to traditional statistics—the set of columns on which distributions are needed is determined early in query compilation. The set of filter predicates defined on the table for the query must be a subset of the *filter_definition* of the statistics object for the statistic to be considered. If multiple such statistics exist, the one with the tightest bounds is used.

Filtered statistics can avoid a common problem in cardinality estimation where estimates become skewed due to data correlation between columns. For example, if you create a table called *CARS,* you might have a column called *MAKE* and a column called *MODEL*. For example, the following table shows that multiple models of cars are made by Ford.

CAR_ID	***MAKE***	***MODEL***
1	Ford	F-150
2	Ford	Taurus
3	BMW	M3

In addition, let's assume that you want to run a query like this:

```
SELECT * FROM CARS WHERE MAKE='Ford' AND MODEL='F-150';
```

When the query processor tries to estimate the selectivity for each condition in an AND clause, it usually assumes that each condition is *independent*. This allows the selectivity of each predicate to be multiplied together to form the total selectivity for the complete WHERE clause. For this example, it would be:

```
2/3 * 1/3 = 2/9
```

The actual selectivity is really 1/3 for this query, because every F-150 is a Ford. This kind of estimation error can be large in some data sets. Detecting statistical correlations like this is a very computationally expensive problem, so the default behavior is to assume independence even though that may introduce some amount of error into the cardinality estimation process.

Filtered Statistics solves this problem by capturing the conditional probability for the *MODEL* column when the *MAKE* value is *Ford.* While using this solution requires a lot of *statistics* objects, it can be effective to fix the most important cases in an application where cardinality estimation error is causing poorly performing plans to be chosen by the Query Optimizer, especially when the WHERE clause has a relatively small number of distinct values.

In additional to the Independence assumption, the Query Optimizer contains other assumptions that are used to both simplify the estimation process and to be consistent in how estimates are made across all operators. Another assumption in the Query Optimizer is the *Uniformity* assumption. This means that if a range of values is being considered but they are not known, then they are assumed to be uniformly distributed over the range in which they exist. For example, if a query has an IN list with different parameters for each value, the values of the parameters are assumed not to be grouped. The final assumption in the Query Optimizer is the *Containment* assumption. This says that if a range of values is being joined with another range of values, then the default assumption is that that query is being asked because those ranges overlap and qualify rows. Without this assumption, many common queries would be underestimated and poor plans would be chosen.

String Statistics

SQL Server 2005 introduced a feature to improve cardinality estimation for strings called *String Statistics* or *Trie Trees*. SQL Server histograms can have up to 200 steps, or unique values, to store information about the overall distribution of a table. While this works well for many numeric types, the string data types often have many more unique values as well as numerous functions that depend more heavily on a deeper statistical understanding of the type, such as *LIKE*. Two hundred unique values is often not sufficient to provide accurate cardinality estimates for strings, and storing lots of strings outside of the table can use a lot of space. Trie trees were created to store a sample of the strings in a column in a space-efficient manner.

The trie tree is not documented, but generally trie trees work as follows:

If we have a column containing the following values:

```
ABC
AAA
ABCDEF
ADAD
BBB
```

The trie tree for this structure is shown in Figure 8-26.

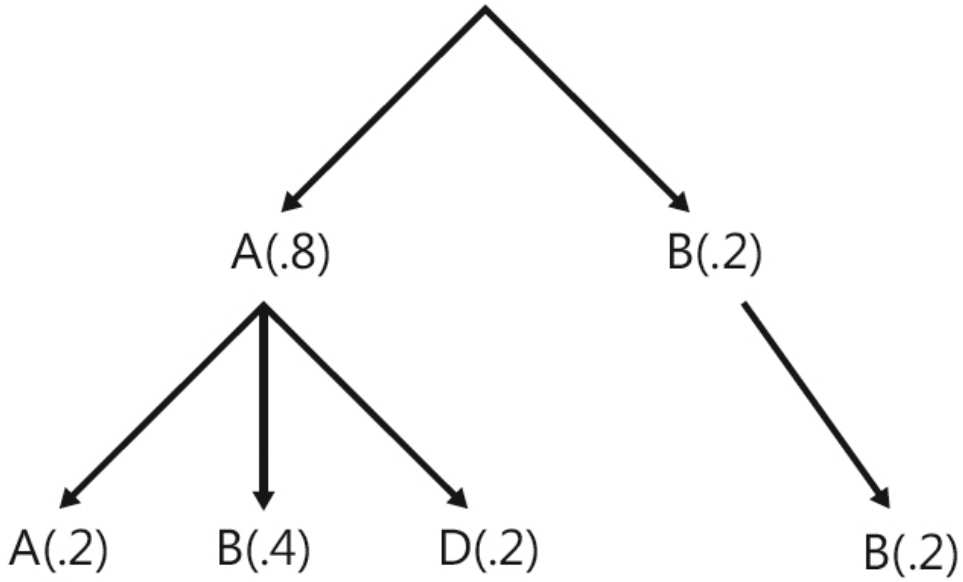


Figure 8-26: Example of a trie tree

SQL Server actually stores a *sample* of the strings in the column, and even this is bound to take up not too much space. SQL Server also has some idea of the relative frequency for each substring listed in the trie tree. Overall, this provides the ability to store far more than 200 unique substrings worth of frequency information.

Cardinality Estimation Details

During optimization, each operator in the query is evaluated to estimate the number of rows that are processed by that operator. This helps the Query Optimizer make proper tradeoffs based on the costs of different query plans. This process is done bottom-up, with the base table cardinalities and statistics being used as input to tree nodes above it. This process continues all the way up the query tree, and the estimated number of rows returned from a query in showplan information is based on this calculation.

Listing 8-7 contains a sample used to explain how the cardinality derivation process works.

Listing 8-7: Cardinality Estimation Sample

```
CREATE TABLE Table3(col1 INT, col2 INT, col3 INT);
GO
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @i INT=0;
WHILE @i< 10000
BEGIN
INSERT INTO Table3(col1, col2, col3) VALUES (@i, @i,@i % 50);
SET @i+=1;
END;
COMMIT TRANSACTION;
GO
SELECT col1, col2 FROM Table3 WHERE col3 < 10;
```

This query is represented in the query processor using the tree shown in Figure 8-27.

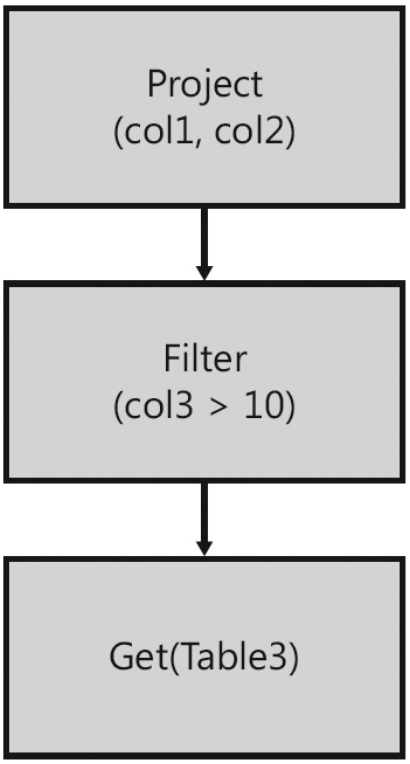


Figure 8-27: Example of a logical query tree for cardinality estimation

For this query, the Filter operator requests statistics on each column participating in the predicate (*col3* in this query). The request is passed down to *Table3,* where an appropriate *statistics* object is created or updated. That *statistics* object is then passed to the filter to determine the *selectivity* of the operator. *Selectivity* is the fraction of rows that are expected to be qualified by the predicate and then returned to the user. Selectivity is used (instead of merely counting the number of matching values in a statistics histogram) to scale the estimate from the sample to the current row count properly, as the current row count may differ from when the *statistics* object was created and the *statistics* object may be over only a sample of the rows.

Once the selectivity for an operator is computed, it is multiplied by the current number of rows for the query. The selectivity of this filter operation is based on the histogram loaded for column *col3*. This can be seen in Figure 8-28.

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	_WA_Sys_00000003_108B795B	Nov 27 2008 10:30AM	10000	10000	50	0	4	NO	NULL	10000

	All density	Average Length	Columns
1	0.02	4	col3

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	0	0	200	0	1
2	1	0	200	0	1
3	2	0	200	0	1
4	3	0	200	0	1
5	4	0	200	0	1
6	5	0	200	0	1
7	6	0	200	0	1
8	7	0	200	0	1
9	8	0	200	0	1
10	9	0	200	0	1
11	10	0	200	0	1

Figure 8-28: Using a histogram to estimate cardinality

I've used a synthetic data distribution for this example to make it easier to follow the computations. Because I've created a distribution on *col3* that is uniformly distributed from 0 to 49, there are 10/50 values less than 10, or 20 percent of the rows. Therefore, the selectivity of this filter in the query is 0.2. So the calculation of the number of rows resulting from the filter is:

(# rows in operator below) * (selectivity of this operator)

10000 * 0.2 = 2000 rows

We can validate this computation by looking at the showplan information for the query shown in Figure 8-29.

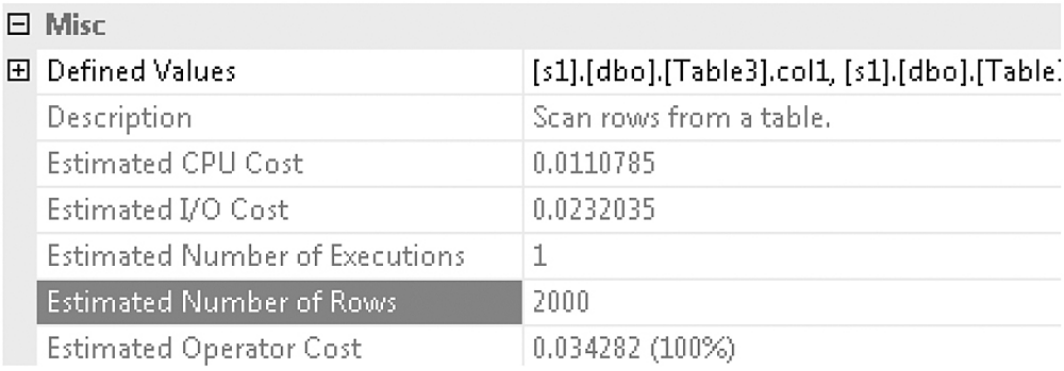


Figure 8-29: Operator row estimate for cardinality example

The estimate for the operator is taken by looking at the histogram, counting the number of sampled rows matching the criteria (in this case, we have 10 histogram steps with 200 equal rows for values that match the filter condition). Then, the number of qualifying rows (2,000) is normalized against the number of rows sampled when the histogram was created (10,000) to create the selectivity for the operator (0.2). This is then multiplied by the current number of rows in the table (10,000) to get the estimated query output cardinality. The cardinality estimation process is continued for any other filter conditions and the results are usually just multiplied to estimate the total selectivity for each condition.

One other interesting aspect of the histogram is *RANGE_ROWS, DISTINCT_RANGE_ROWS,* and *AVG_RANGE_ROWS.* Because histograms are limited to 200 steps, some values being queried may not be represented in the histogram steps. These values are represented in the *RANGE* values, and they are counts of rows between the step values. For query conditions that do not match one of the equal (EQ) rows in the histogram, values in the range are assumed to be uniformly distributed over the domain between the two bounding histogram steps. The fraction is determined from this assumption and used to generate the selectivity, as in the previous examples.

Although most operators work using a mechanism similar to Filter, some other operators need additional mechanisms to make good cardinality estimates. For example, *GROUP BY* actually doesn't try to determine which slices of a histogram should be used to estimate the selectivity of the operator. Instead, it needs to determine the number of unique values over a set of columns, as can be seen in Figure 8-30. This information can be estimated by looking at the histogram, but there is another mechanism in the *statistics* object to help perform this calculation quickly. The *density* information is stored in the histogram in the second result set, and in this case, it is 0.02 for *col3*.

```
SELECT COUNT(*) FROM Table3 GROUP BY col3;
```

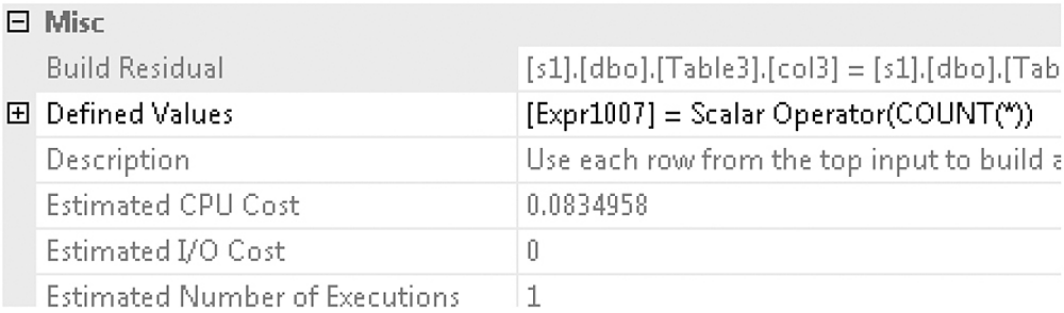


Figure 8-30: GROUP BY cardinality estimate

This is a representation of the average number of duplicates for any value in the table. In other words, this tells us how to compute the number of groups using the total number of rows. For this simple *GROUP BY* query, the estimate of rows is (1/0.02)*(10,000/10,000) = 50, which matches the number of groups we would expect from our creation script:

```
GROUP BY Selectivity = (1/density)
GROUP BY Card. Estimate = (Input operator) * (selectivity)
```

When a multicolumn statistics object is created, it computes density information for the sets of columns being evaluated in the order of the *statistics* object. So a *statistics* object created on (*col1, col2, col3*) has density information stored for ((*col1*), (*col1, col2*), and (*col1, col2, col3*)). This can be used to compute the cardinality estimate for a query over that table doing *GROUP BY col1, GROUP BY col1, col2*, or *GROUP BY col1, col2, col3*.

We can see this computation in the results of *DBCC SHOW_STATISTICS* in Figure 8-31.

```
CREATE TABLE Table4(col1 int, col2 int, col3 int)
GO
DECLARE @i int=0
WHILE @i< 10000
BEGIN
INSERT INTO Table4(col1, col2, col3) VALUES (@i % 5, @i % 10,@i % 50);
SET @i+=1
END
CREATE STATISTICS s1 on Table4(col1, col2, col3)
DBCC SHOW_STATISTICS (Table4, s1)
```

	All density	Average Length	Columns
1	0.2	4	col1
2	0.1	8	col1, col2
3	0.02	12	col1, col2, col3

Figure 8-31: Multicolumn density information

Note SQL Server does not automatically create statistics for multicolumn cases like this except in index creation, so we need to create the statistics object manually for this example.

Multicolumn density information is important because it captures correlation data between columns in the same table. By default, if every column were assumed to be completely independent, then one would expect a large number of different groups to be returned because each column added to the grouping columns would add more and more uniqueness (and less and less selectivity for the *GROUP BY* operator). However, in this case, we see that the selectivity of the *GROUP BY* is the same as in our previous example—50 groups. The data captured in the multicolumn density can be used to get this cardinality estimation to be more accurate.

If I create a similar table with random data in the first two columns, the density looks quite different, as shown in Figure 8-32.

	All density	Average Length	Columns
1	0.01	4	col1
2	0.0001579031	8	col1, col2
3	0.0001009387	12	col1, col2, col3

Figure 8-32: Multicolumn density for random data distribution

This would imply that every combination of *col1, col2*, and *col3* are actually unique in that case. By examining the various inputs into the cardinality estimation process, it is possible to determine whether the plan used reasonable information during the compilation process.

There are many, many more details to cardinality estimation than can be covered in this chapter. Most of the details change somewhat from release to release, and most of the details are not exposed or documented enough to make it useful to try to follow the exact computation. It is still very useful to understand the statistics and cardinality estimation mechanism so that you can perform plan debugging and hinting (explained later in this chapter).

Limitations

The cardinality estimation of SQL Server is usually very good. Unfortunately, it is very difficult to make a model that is perfect for every query for all applications. While most of these are internal details, some of them are interesting to know

about so that you can understand that the calculations explained earlier in this section do not work perfectly in every query.

- **Multiple predicates in an operator** The selectivities of multiple predicates are multiplied during cardinality estimation to determine the resulting estimate for the whole operator. This means that the predicates are assumed to be statistically independent. In practice, most data has some statistical dependencies between columns. As the number of predicates in the query increases, the Query Optimizer actually does not directly multiply all the selectivities and assumes that these different predicates are related. So the selectivity of an operator with many predicates may be greater than you might expect.
- **Deep Query Trees** The process of tree-based cardinality estimation is good, but it also means that any errors lower in the query tree are magnified as the calculation proceeds higher up the query tree to more and more operators. Eventually, the error introduced in all these computations overwhelms the value of using histograms to compute cardinality estimates. As a result, very deep query trees eventually stop using histograms for the higher portions of the query tree and may use simpler heuristics to make cardinality estimates to avoid assuming information on data that is very likely to be invalid.
- **Less common operators** The Query Optimizer has many operators. Most of the common operators have extremely deep support developed over multiple versions of the product. Some of the lesser-used operators, however, don't necessarily have the same depth of support for every single scenario. So if you are using an infrequent operator or one that has only recently been introduced, it may not provide cardinality estimates that are as good as the most core operators. In these cases, it is worth double-checking the estimates using *SET STATISTICS PROFILE ON* to see if the estimates are close to what is expected. In many cases where the estimates are incorrect, the impact is often mitigated because specialized operators do not always have many plan choices and the impact of an error in cardinality estimation may be reduced.

Costing

The process of estimating cardinality is done using the logical query trees. *Costing* is the process of determining how much time each potential plan choice will take to run, and it is done separately for each physical plan considered. Given that the Query Optimizer considers multiple different physical plans that return the same results, this makes sense. Costing is the component that picks between hash joins and loops joins or between one join order and another.

The idea behind costing is actually quite simple. Using the cardinality estimates and some additional information about the average and maximum width of each column in a table, it is able to determine how many rows fit on each database page. This value is then translated into a number of disk reads that a query requires to complete. The total costs for each operator are then added to determine the total query cost, and the Query Optimizer is able to select the fastest (lowest-cost) query plan from the set of considered plans during optimization.

In practice, costing is not this simple. There is a cost difference between sequential I/Os, where disk blocks are stored sequentially on disk and therefore do not require waiting to move the disk head to a new track or even waiting for a complete rotation of the disk platter, and random I/Os, where neither of these conditions are guaranteed to be true. In addition, some queries are large enough that the data can be read into memory and read multiple times during a query. These additional reads are often going to be able to read the page from the memory-based page buffer pool, avoiding the need to read from disk at all. Even further, some queries may take more memory than is available in the server for the query— in this case, the costing component needs to determine that some pages get evicted from the buffer pool and must be reread, either randomly or sequentially. The Optimizer uses logic to consider all these conditions, and the process of determining the actual cost for an operator can take awhile to calculate. All these considerations help make sure that SQL Server does the best job possible to select a good query plan for each query.

To make the Query Optimizer more consistent, the development team used several assumptions when creating the costing model. First, a query is assumed to start with a cold cache. This means that the query processor assumes that each initial I/O for a query requires reading from disk. In a very small number of cases (usually small, OLTP queries), this may cause the Query Optimizer to pick a slightly slower plan that optimizes for the number of initial I/Os required to complete the query. The cold-cache assumption is a simplification that allows the query processor to generate plans more consistently, but it is a (small) difference between the mathematical model used to compare plans and reality. Second, random I/Os are assumed to be evenly dispersed over the set of pages in a table or index. If a non-indexed based table (a heap) has 100 disk pages and the query is doing 100 random bookmark-based lookups into the heap, the Query Optimizer assumes that 100 random I/Os occur in the query because it assumes that each target row is on a separate page. Like the statistical column correlation example earlier in the chapter, this assumption also does not always hold. The actual set of rows could be clustered physically on the same pages (perhaps they were all inserted at the same time and thus ended up on adjacent pages), and it may only require five I/Os to read the rows of interest. In this case, the Query Optimizer would

overcost this query. This also rarely happens, but it is valuable to understand that the mathematical model used for costing is just that—a model. In the rare cases when the model does not work properly, query hints can be used to help force a different query plan.

The Query Optimizer has other assumptions built into its costing model. One assumption relates to how the client reads the query results. Costing assumes every query reads every row in the query result. However, some clients read only a few rows and then close the query. For example, if you are using an application that shows you pages of rows on a screen at a time, then that application may read 40 rows even though the original query may have returned 10,000 rows. If the Query Optimizer knows the number of rows that the user will consume, then it can optimize for the number in the plan selection process to pick a faster plan. Typically, this causes the Query Optimizer to switch from using operators such as hash join (which has a larger startup cost at the beginning of a query) to nested loops joins (which have a lower startup cost but a higher per-row cost).

SQL Server exposes a hint called FAST N for just this case. If a user typically only reads a subset of the rows in a query, then it can pass OPTION (FAST N) to the query to tell the Query Optimizer to cost the query for returning *N* rows instead of the whole result set. Listing 8-8 contains an example to demonstrate the FAST N hint, which selects a hash join without the FAST N hint. Figure 8-33 shows that a loop join is picked when the hint is applied.

Listing 8-8: FAST N Example

```
CREATE TABLE A(col1 INT);
CREATE CLUSTERED INDEX i1 ON A(col1);
GO
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @i INT=0;
WHILE @i < 10000
BEGIN
INSERT INTO A(col1) VALUES (@i);
SET @i+=1;
END;
COMMIT TRANSACTION;
GO
SELECT A1.* FROM A as A1 INNER JOIN A as A2 ON A1.col1=A2.col1;
SELECT A1.* FROM A as A1 INNER JOIN A as A2 ON A1.col1=A2.col1 OPTION (FAST 1);
```

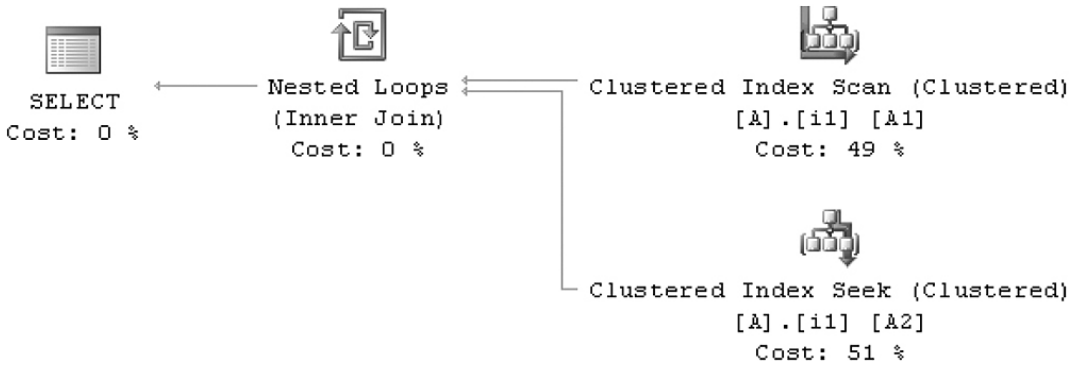


Figure 8-33: Loops join plan (with FAST 1 hint)

Index Selection

Index selection is one of the most important aspects of query optimization. The basic idea behind index matching is to take predicates from a WHERE clause, join condition, or other limiting operation in a query and to convert that operation that can be performed against an index. Two basic operations can be performed against an index:

- Seek (for a single value or a range of values on the index key)
- Scan the index (forwards or backwards)

For Seek, the initial operation starts at the root of a B+ tree and navigates down the tree to a desired location in the index based on the index keys. Once completed, the query processor can iterate over all rows that match the predicate or until

the last value in the range is found. Because leaves in a B+ tree are linked in SQL Server, it is possible to scan rows in order using this structure once the intermediate B+ tree nodes have been traversed.

The job of the Query Optimizer is to figure out which predicates can be applied to the index to return rows as quickly as possible. Some predicates can be applied to an index, while others cannot. For example, the query *SELECT col1, PKcol FROM MyTable WHERE col1=2* has one predicate in the form of <column> = <constant>. This pattern can be matched to a seek operation if there is an index on that column. The resulting alternative that is generated is to perform a seek against the nonclustered index and to return the rows that match, if any. Figure 8-34 demonstrates a basic seek plan generated by the Query Optimizer.

```
CREATE TABLE idxtest2(col2 INT, col3 INT, col4 INT);
CREATE INDEX i2 ON idxtest2(col2, col3);

SELECT col2, col3 FROM idxtest2 WHERE col2=5
```

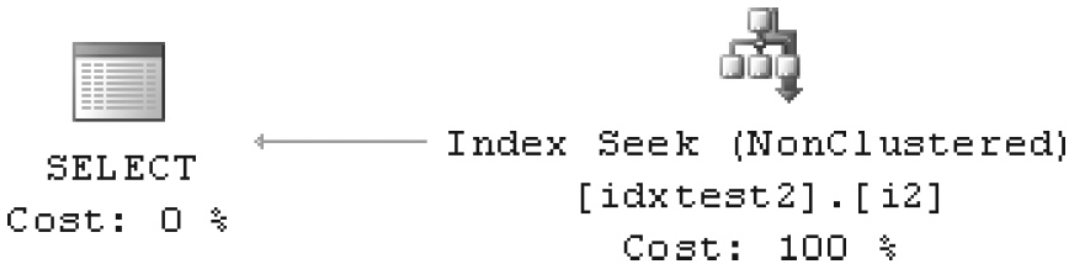


Figure 8-34: Index seek plan

The Query Optimizer can also apply compound predicates against multicolumn indexes as long as the operation can be converted into starting and ending index keys. Figure 8-35 shows a multicolumn seek plan, and you can see the predicates used if you look at the properties for this operator in Management Studio.

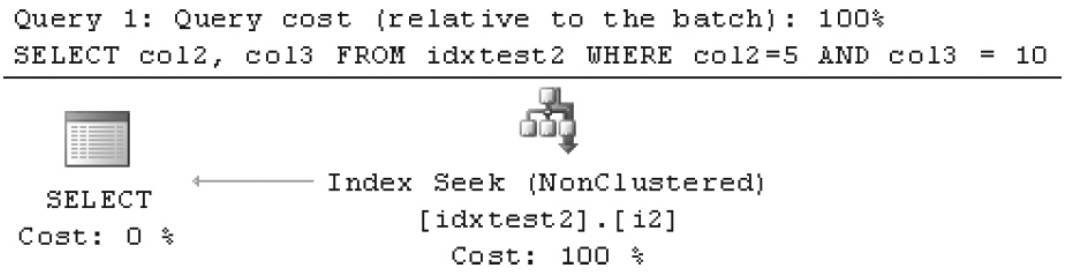


Figure 8-35: Multicolumn index seek plan

Predicates that can be converted into an index operation are often called *sarg*able, or " Search-ARGument-able." This means that the form of the predicate can be converted into an index operation. Predicates that cannot ever match or do not match the selected index are called *non-sargable predicates*. Predicates that are non-sargable would be applied after any index seek or range scan operations so that the query can return rows that match all predicates. Making things somewhat confusing is that SQL Server usually evaluates non-sargable predicates within the seek/scan operator in the query tree. This is a performance optimization—if this were not done, the series of steps performed would be as follows:

1. Seek Operator: Seek to a key in an index's B+ tree.
2. Latch the page.
3. Read the row.
4. Release the latch on the page.
5. Return the row to the filter operator.
6. Filter: evaluate the non-sargable predicate against the row. If it qualifies, pass the row to the parent operator. Otherwise, go to step 2 to get the next candidate row.

This is slower than optimal because returning the row to a different operator requires loading in a different set of instructions and data to the CPU. By keeping the logic in one place, the overall CPU cost of evaluating the query goes

down. The actual operation in SQL Server looks like this:

1. Seek Operator: Seek to a key in an index's B+ tree.
2. Latch the page.
3. Read the row.
4. Apply the non-sargable predicate filter. If the row does not pass the filter, go to step 3. Otherwise, continue to step 5.
5. Release the latch on the page.
6. Return the row.

This is called *pushing non-sargable predicates* (the predicate is pushed from the filter into the seek/scan). It is a physical optimization, but it can show up in queries that process many rows.

Not all predicates can be evaluated in the seek/scan operator. Because the latch operation prevents other users from even looking at a page in the system, this optimization is reserved for predicates that are very cheap to perform. This is called *non-pushable, non-sargable predicates*. Examples include:

- Predicates on large objects (including *varbinary(max), varchar(max), nvarchar(max)*)
- CLR functions
- Some T-SQL functions

Predicate sargability is an important consideration in database application design. One reason systems can perform poorly is that the application against the database is written in such a way as to make predicates non-sargable. In many cases, this is avoidable if the issue is identified early enough, and fixing this one issue can sometimes increase database application performance by an order of magnitude.

SQL Server considers many formulations when trying to apply indexes against sargable predicates in a query. For AND conditions (WHERE *col1*=5 AND *col2*=6 AND …), SQL Server tries to do the following:

1. Given a list of required seek equality columns, seek inequality columns, and columns needed to satisfy the query but without predicates, first attempt to find an index that exactly matches the request. If such an index exists, use it.
2. Try to find a set of indexes to satisfy the equality conditions and perform an inner join for all such indexes.
3. If step 2 did not cover all required columns, consider joins with any other indexes based on the set of columns in the indexes included so far in the solution.
4. Finally, perform a join back to the base table to get any remaining columns.

In all cases, the costs of each solution are considered and the solution is only returned if it is believed to be least-cost. So, a solution that joins many indexes together will only be used if it is believed to be cheaper than a scan of all rows in the base table. Second, this algorithm is performed locally in the query tree. Even if the Query Optimizer generates a specific alternative using this process, it may not ultimately be part of the final query plan. Costing is used to determine the cheapest, complete query plan. So, this is not a rule-based mechanism for selecting indexes. It is a heuristic that is part of a broader costing infrastructure to help choose efficient query plans.

Filtered Indexes

SQL Server 2008 introduces the ability to create indexes with simple predicates that restrict the set of rows included in the index. On first glance, this feature is a subset of the functionality already contained in indexed views. Nevertheless, there are good reasons for this feature to exist. First, indexed views are more expensive to use and maintain. Second, the matching capability of the Indexed View feature is not supported in all editions of SQL Server. Third, a number of different SQL Server users had scenarios that were just slightly more complex than the regular index feature and therefore, they were not really interested in moving to a full indexed view solution. So, although indexed views are still a very useful feature, they tend to be more useful for the more classical relational query precomputation scenarios.

Filtered Indexes are created using a new WHERE clause on a *CREATE INDEX* statement.

Listing 8-9 demonstrates how to create an index and how it can be used in a query. Figures 8-36 and 8-37 show the

resulting query plans for a query where the filtered index is covering and when it is not, respectively.

Listing 8-9: Filtered Index Example

```
CREATE TABLE testfilter1(col1 INT, col2 INT);
go
DECLARE @i INT=0;
SET NOCOUNT ON;
BEGIN TRANSACTION;
WHILE @i < 40000
BEGIN
INSERT INTO testfilter1(col1, col2) VALUES (rand()*1000, rand()*1000);
SET @i+=1;
END;
COMMIT TRANSACTION;
go

CREATE INDEX i1 ON testfilter1(col2) WHERE col2 > 800;

SELECT col2 FROM testfilter1 WHERE col2 > 800;
SELECT col2 FROM testfilter1 WHERE col2 > 799;
```

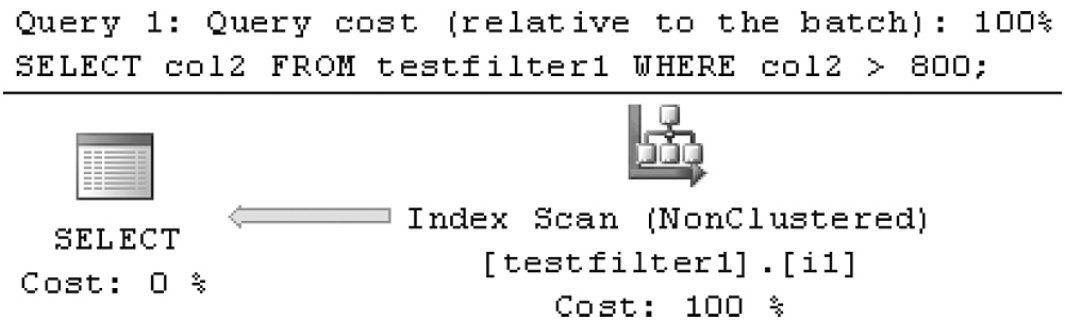


Figure 8-36: Filtered index used in query plan

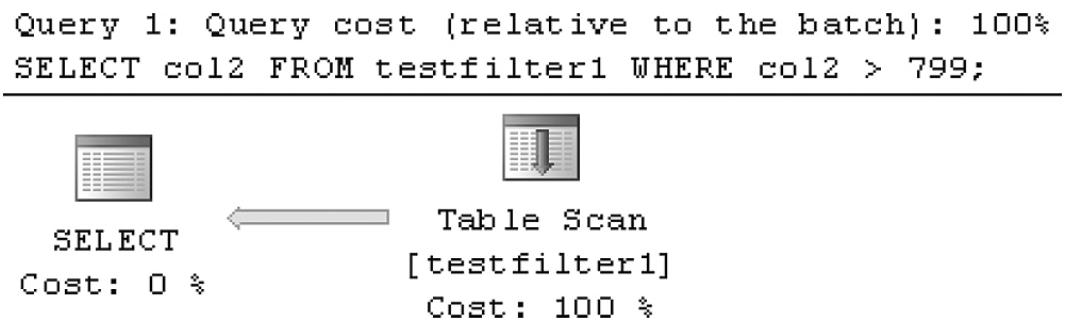


Figure 8-37: Filtered index not used due to noncovering filter condition

The cost of the first select query is 0.0141293, whereas the second query has an estimated cost of 0.112467. The filtered index benefits from having fewer rows and is also narrower than the base table, so it has fewer pages as well. When you know specific constraints that are used on queries with large tables where space is an issue, this kind of index can be quite useful.

SQL Server imposes a number of restrictions on the scalar constructs that can be used to formulate the filter in the *CREATE INDEX* command. These are largely based on what the Query Optimizer's domain property framework can use easily when matching indexes. As a result, some of the more complex pieces of the system are not supported in this release because there is no way to match these indexes efficiently.

Several scenarios can be handled by filtered indexes:

- Not all data fits easily into the relational database model with a small, fixed set of columns that are set for every row. Often, some fields are used only occasionally, resulting in many NULL entries for that column. A traditional index stores

a lot of NULLs and wastes a lot of storage space. Updates to the table have to maintain this index for every row.

- If you are querying a table with a small number of distinct values and are using a multicolumn predicate where some of the elements are fixed, you can create a filtered index to speed up this specific query. This might be useful for a regular report run only for your boss—it speeds up a small set of queries while not slowing down updates as much for everyone else.
- As shown in the original example, the index can be used when there is a known query condition on an expensive query on a large table.

Indexed Views

Traditional, non-indexed views have been used for goals such as simplifying SQL queries, abstracting data models from user models, and enforcing user security. From an optimization perspective, SQL Server does not do much with these views because they are expanded, or in-lined, before optimization begins. This gives the Query Optimizer opportunities to optimize queries globally, but it also makes it difficult for the Query Optimizer to consider plans that perform the view evaluation first, then process the rest of the query. Arbitrary tree matching is a computationally complex problem, and the feature set of views is too large to perform this operation efficiently.

Note Matching of indexed views is supported only in SQL Server 2008 Enterprise Edition.

The Indexed Views feature allows SQL Server to expose some of the benefits of view materialization while retaining the benefits of global reasoning about query operations. SQL Server exposes a *CREATE INDEX* command on views that creates a materialized form of the query result. The resulting structure is physically identical to a table with a clustered index. Nonclustered indexes also are supported on this structure. The Query Optimizer can use this structure to return results more efficiently to the user. The Query Optimizer contains logic to use this index both in cases when the original query text referenced the view explicitly as well as in cases when the user submits a query that uses the same components as the view (in any equivalent order). Actually, the query processor expands indexed views early in the query pipeline and always uses the same matching code for both cases. The WITH(NOEXPAND) hint tells the query processor not to expand the view definition. Listing 8-10 contains an example with three different paths to get SQL Server to match the view. The plans for the matches are visible in Figures 8-38, 8-39, and 8-40.

Listing 8-10: Indexed View Matching Examples

```
-- Create two tables for use in our indexed view
CREATE TABLE table1(id INT PRIMARY KEY, submitdate DATETIME, comment NVARCHAR(200));
CREATE TABLE table2(id INT PRIMARY KEY IDENTITY, commentid INT, product NVARCHAR(200));
GO
-- submit some data into each table
INSERT INTO table1(id, submitdate, comment) VALUES (1, '2008-08-21', 'Conor Loves Indexed
Views');
INSERT INTO table2(commentid, product) VALUES (1, 'SQL Server 2008');
GO
-- create a view over the two tables
CREATE VIEW dbo.v1 WITH SCHEMABINDING AS
SELECT t1.id, t1.submitdate, t1.comment, t2.product FROM dbo.table1 t1 INNER JOIN dbo.table2
t2 ON t1.id=t2.commentid;
go
-- indexed the view
CREATE UNIQUE CLUSTERED INDEX i1 ON v1(id);

-- query the view directly --> matches
SELECT * FROM dbo.v1;
-- query the statement used in the view definition --> matches as well
SELECT t1.id, t1.submitdate, t1.comment, t2.product
FROM dbo.table1 t1 INNER JOIN dbo.table2 t2
    ON t1.id=t2.commentid;
-- query a logically equivalent statement used in the view definition that
-- is written differently --> matches as well
SELECT t1.id, t1.submitdate, t1.comment, t2.product
FROM dbo.table2 t2 INNER JOIN dbo.table1 t1 ON t2.commentid=t1.id;
```

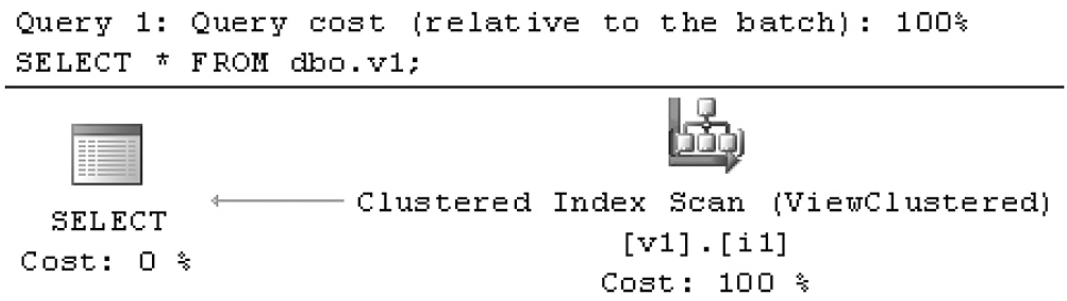


Figure 8-38: A direct reference match of an indexed view

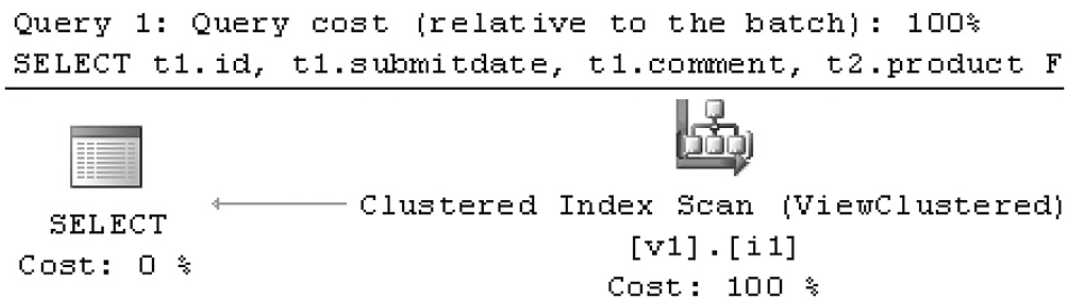


Figure 8-39: An Indexed View match when the query is a match to the view definition

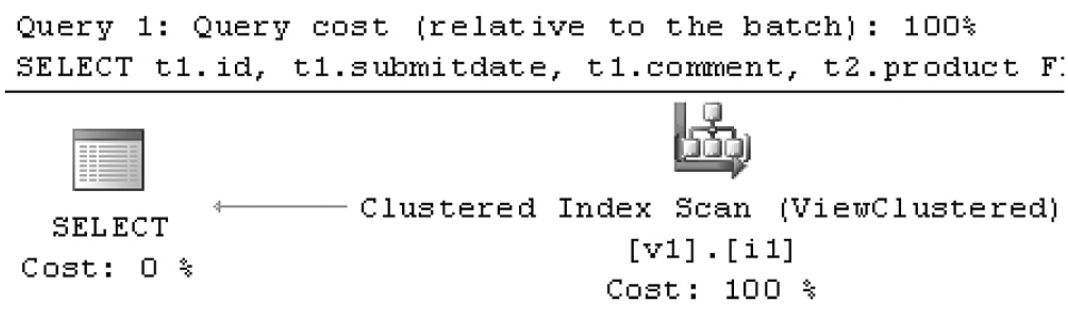


Figure 8-40: An Indexed View match when the query is not an exact match to the view definition

There are cases when the Query Optimizer does not match the view. First, remember that indexed views are inserted into the Memo and evaluated against other plan choices. While they are often the best plan choice, this is not always the case. In Listing 8-11, the Query Optimizer can detect logical contradictions between the view definition and the query that references the view. Figure 8-41 shows the query plan that directly references the base table instead of the view.

Listing 8-11: Example When an Index View Is Not Matched

```
CREATE TABLE table3(col1 INT PRIMARY KEY IDENTITY, col2 INT);
INSERT INTO table3(col2) VALUES (10);
INSERT INTO table3(col2) VALUES (20);
INSERT INTO table3(col2) VALUES (30);
GO
-- create a view that returns values of col2 > 20
CREATE VIEW dbo.v2 WITH SCHEMABINDING AS
SELECT t3.col1, t3.col2 FROM dbo.table3 t3 WHERE t3.col2 > 20;
GO
-- materialize the view
CREATE UNIQUE CLUSTERED INDEX i1 ON v2(col1);
GO

-- now query the view and filter the results to have col2 values equal to 10.
-- The optimizer can detect this is a contradition and avoid matching the indexed view
-- (the trivial plan feature can "block" this optimization)
SELECT * FROM dbo.v2 WHERE col2 = CONVERT(INT, 10);
```

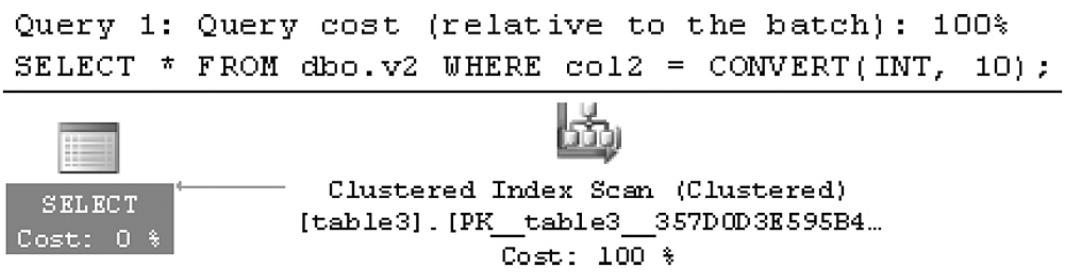


Figure 8-41: A query plan when Indexed View is not matched

Note The predicate in this example is *[v1].[dbo].[table3].[col2] as [t3].[col2]=[@1] AND [v1].[dbo]. [table3].[col2] as [t3]. [col2]>(20)*. While I have tried to make the examples in this chapter as simple as possible, the Query Optimizer uses logic here to detect that I have made this query example too simple. As a result, it has treated it like a trivial plan and auto-parameterized it for use by all future queries like this one that vary only by the constant (10). Although the intricacies of trivial plan are not formally documented and are subject to change each release, Figure 8-42 shows what could happen when you modify the query slightly to avoid the trivial plan feature (in my case, I used a query hint, but that is not shown in the code).

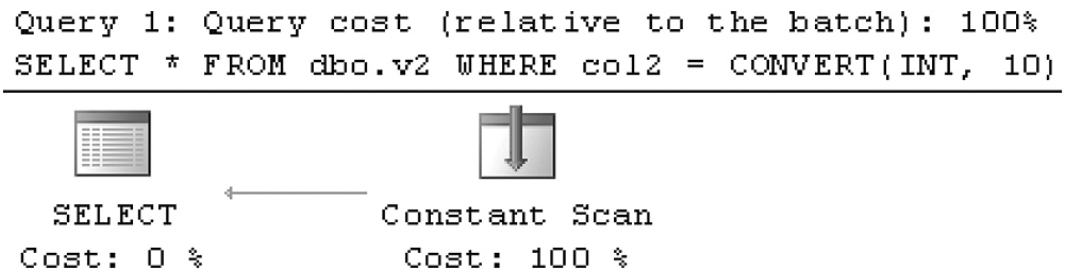


Figure 8-42: A Constant Scan plan due to non-trivial plan contradiction detection

This is a zero-row scan because the Query Optimizer recognizes that *col2* = 10 and *col2* > 20 never return rows. This query plan doesn't even try to scan *table3* or *v2*.

Tip Unfortunately, there are also some cases where the Query Optimizer does not recognize an indexed view even when it would be a good plan choice. Often, these cases deal with complex interactions between high-level features within the query processor (such as computed column matching and the algorithm to explore join orders). Although SQL Server does provide some information through warnings and showplans that can help you see the behaviors of the system at this level, it requires a lot of internal knowledge to understand fully. If you happen to find yourself in a case where you believe that the indexed view should match but does not, then consider the WITH (NOEXPAND) hint to force the query processor to pick that indexed view. This usually is enough to get the plan to include the indexed view.

SQL Server also supports matching indexed views in cases beyond exact matches of the query text to the view definition. It also supports using an indexed view for inexact matches where the definition of the view is broader than the query submitted by the user. SQL Server then applies residual filters, projections (columns in the select list), and even aggregates to use the view as a partial precomputation of the query result.

Listing 8-12 demonstrates view matching for both filter and projection residuals. It creates a view that has more rows and more columns than our final query, but the indexed view is still matched by the Query Optimizer. The resulting query plan is shown in Figure 8-43.

Listing 8-12: Indexed View Matching Example (A Subset of Rows and Columns)

```
-- base table
CREATE TABLE basetbl1 (col1 INT, col2 INT, col3 BINARY(4000));
CREATE UNIQUE CLUSTERED INDEX i1 ON basetbl1(col1);
GO
-- populate base table
SET NOCOUNT ON;
DECLARE @i INT =0;
WHILE @i < 50000
BEGIN
```

```
INSERT INTO basetbl1(col1, col2) VALUES (@i, 50000-@i);
SET @i+=1;
END;
GO
-- create a view over the 2 integer columns
CREATE VIEW dbo.v2 WITH SCHEMABINDING AS
SELECT col1, col2 FROM dbo.basetbl1;
GO
-- index that on col2 (base table is only indexed on col1)
CREATE UNIQUE CLUSTERED INDEX iv1 on dbo.v2(col2);

-- the indexed view still matches for both a restricted
-- column set and a restricted row set
SELECT col1 FROM dbo.basetbl1 WHERE col2 > 2500;
```

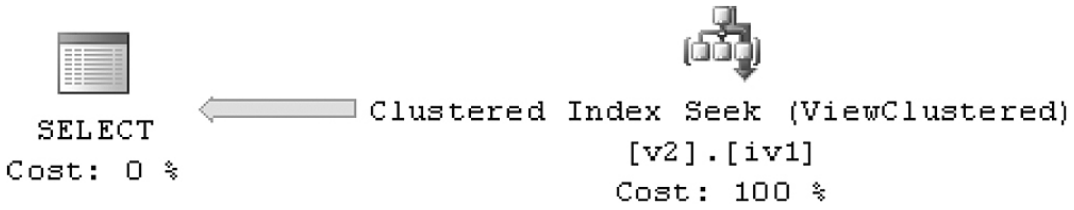


Figure 8-43: An indexed view matched for a subset of rows and columns

The projection is not explicitly listed as a separate Compute Scalar operator in this query because SQL Server 2008 has special logic to remove projections that do not compute an expression. The filter operator in the index matching code is translated into an index seek against the view. If we modify the query to compute an expression, Figure 8-44 demonstrates the residual Compute Scalar added to the plan:

```
SELECT col1 + 1 FROM dbo.basetbl1 WHERE col2 > 2500 AND col1 > 10;
```

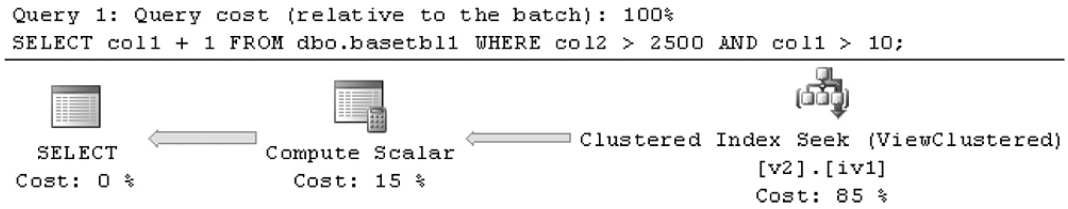


Figure 8-44: Compute Scalar, only needed when computing new values

Like all options considered by the Query Optimizer, indexed view alternatives are generated and stored in the Memo and are compared using costing equations against other possible plans. Alternatives including partial matches cost the residual operations as well, and this means that an indexed-view plan can be generated but not picked when the Query Optimizer considers other plans to have lower costs.

Indexed views are maintained as part of the update processing for tables on which the view is based. This makes sure that the view provides a consistent result if it is selected by the Query Optimizer for any query plan. Some query operations are incompatible with this design guarantee. As a result, SQL Server places some restrictions on the set of supported constructs in indexed views to make sure that the view can be created, matched, and updated as efficiently as possible. The description of the restrictions in *SQL Server Books Online* is very long and detailed, and this can make it very difficult to understand the higher-level rules.

For updating indexed views, the core question behind the restrictions is “Can the query processor compute the necessary changes to the Indexed View clustered and nonclustered indexes without having to recompute the whole indexed view?” If so, the query processor can perform these changes efficiently as part of the maintenance of the base tables that are referenced in the view. This property is relatively easy for filters, projections (compute scalar), and inner joins on keys. Operators that destroy or create data are more difficult to maintain, so often these are restricted from use in indexed views.

How indexed views are represented in update plans is discussed in the section entitled “Updates,” later in this chapter.

Partitioned Tables

As SQL Server is used to store more and more data, management of very large databases becomes a bigger concern for DBAs. First, the time to perform operations like an index rebuild grows with the data size, and eventually this can affect system availability. Second, the size of large tables makes performing operations difficult because the system is often strained for resources, such as temp space, log space, and physical memory. Table and index partitioning can help you manage large databases better and minimize downtime.

Physically, partitioned tables and indexes are really *N* tables or indexes that store a fraction of the rows. When comparing this to their nonpartitioned equivalents, the difference in the plan is often that the partitioned case requires iterating over a list of tables or a list of indexes to return all the rows. In SQL Server 2005, this was represented using an APPLY operator, which is essentially a nested loops join. In the 2005 representation, a special table of partition IDs was passed in as parameters to the query execution component in a join to iterate over each partition. While this works well in most cases, there are some important scenarios that didn't work well with this model. For example, there is a restriction in parallel query plans that requires that the parallel table or index scan feature (where multiple threads read rows from a table at once to improve performance) did not work on the inner side of a nested loops join, and this was not possible to fix before SQL Server 2005 shipped. Unfortunately, this is the majority case for table partitioning. In addition, the APPLY representation enabled join collocation, where two tables partitioned in the same way can be joined very efficiently. Unfortunately, this turned out to be less common in practice than was foreseen when the feature was originally designed. For reasons like this, the representation was refined further in the 2008 version of the product.

SQL Server 2008 represents partitioning in most cases by storing the partitions within the operator that accesses the partitioned table or index. This provides a number of benefits, like enabling parallel scans to work properly. It also removed a number of other differences between partitioned and nonpartitioned cases in the Query Optimizer that manifested themselves as missed performance optimizations. Hopefully this makes it easier to deploy partitioning in applications that started out nonpartitioned.

Listing 8-13 contains the example to show this new design. Figure 8-45 contains the resulting query plan for SQL Server 2008 over partitioned tables.

Listing 8-13: SQL Server 2008 Partitioning Example—No Apply Needed

```
CREATE PARTITION FUNCTION pf2008(date) AS RANGE RIGHT
       FOR VALUES ('2008-10-01', '2008-11-01', '2008-12-01');

CREATE PARTITION SCHEME ps2008 AS PARTITION pf2008 ALL TO ([PRIMARY]);
CREATE TABLE ptnsales(saledate DATE, salesperson INT, amount MONEY) ON ps2008(saledate);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2008-10-20', 1, 250.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2008-11-05', 2, 129.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2008-12-23', 2, 98.00);
INSERT INTO ptnsales (saledate, salesperson, amount) VALUES ('2008-10-3', 1, 450.00);

SELECT * FROM ptnsales WHERE (saledate) NOT BETWEEN '2008-11-01' AND '2008-11-30';
```

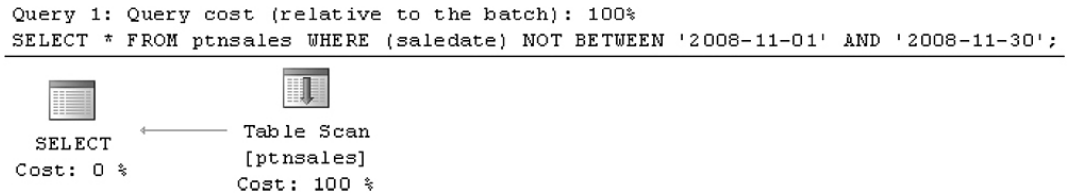


Figure 8-45: Query plan for the new SQL Server 2008 partitioning model

You can see that the base case doesn't require an extra join with a Constant Scan. This makes the query plans look like the nonpartitioned cases more often, which should make it easier to understand the query plans.

One benefit of this model is that it is now possible to get parallel scans over partitioned tables. The following example creates a large partitioned table and then performs a *COUNT(*)* operation that generates a parallel scan. In SQL Server, some aggregate functions can be split into two parts, with one part executed in the same thread as the table. This can speed up execution time in large queries and minimize the number of rows that need to be passed from thread to thread. Listing 8-14 demonstrates how SQL Server 2008 now generates parallel scans over partitioned tables to compute aggregates. Figure 8-46 contains the resulting query plan.

Listing 8-14: Partitioned Parallel Scan Example—SQL Server 2008

```
CREATE PARTITION FUNCTION pfparallel(INT) AS RANGE RIGHT FOR VALUES (100, 200, 300);
CREATE PARTITION SCHEME psparallel AS PARTITION pfparallel ALL TO ([PRIMARY]);
GO
CREATE TABLE testscan(randomnum INT, value INT, data BINARY(3000)) ON psparallel(randomnum);
GO
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @i INT=0;
WHILE @i < 100000
BEGIN
INSERT INTO testscan(randomnum, value) VALUES (rand()*400, @i);
SET @i+=1;
END;
COMMIT TRANSACTION;
GO
-- now let's demonstrate a parallel scan over a partitioned table in SQL Server 2008
SELECT COUNT(*) FROM testscan;
```

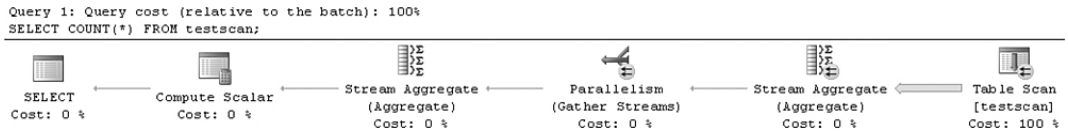


Figure 8-46: Parallel scan on partitioned tables in SQL Server 2008

SQL Server 2005 had limitations on how parallel queries could be executed against partitioned tables. The use of the APPLY operator to scan each partition interacted poorly with some other restrictions in the system to allow SQL Server 2005 to run only one thread per partition. Although this allowed the query to run in parallel when scanning many partitions, this model did not work well when the query accessed a single partition. When accessing a single partition, only one thread could access the partition, essentially ignoring the Parallel Scan feature. Unfortunately, one of the core reasons for SQL Server range partitioning is to access the most current partition in a date range. In addition, the APPLY model also made it difficult to handle partition skew (where one partition is much larger than others) efficiently. While SQL Server 2005 would consider the size of the largest partition when costing a query using this pattern, it still has one thread finishing later than the other threads.

The Query Optimizer has improved the end-to-end experience in partitioned table plan generation. The ability to represent partitioned table access in the same manner as nonpartitioned access guarantees that the performance differences between partitioned and nonpartitioned tables are minimized. Specifically, the set of considered parallel plan options is much more consistent. The query execution component can dynamically adjust between using one thread per partition and using multiple threads per partition, which should allocate threads to finish processing a query more efficiently.

In SQL Server 2008, join collocation is still represented using the apply/nested loops join, but other cases use the traditional representation. This works with other features within the query processor to guarantee that they behave the same as nonpartitioned tables. The following example builds upon the last example to demonstrate that joining two tables with the same partitioning scheme can be done using the collocated join technique. The scenarios for this remain the same as in SQL Server 2005—cases when you want to join two partitioned tables or indexes together. Often, this would be a fact table and a large dimension table index that is partitioned in the same manner as the fact table. Figure 8-47 shows a per-partitioned join example when the original SQL Server 2005 partitioning logic is still visible.

```
-- SQL Server 2008 join collocation still uses the constant scan + apply model
SELECT * FROM testscan t1 INNER JOIN testscan t2 ON t1.randomnum=t2.randomnum;
```

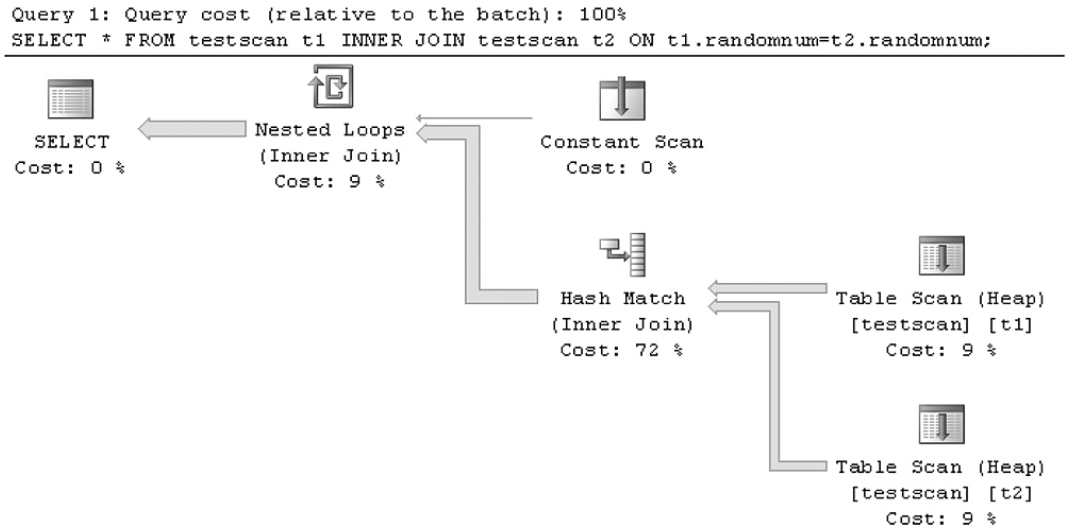


Figure 8-47: Query plan for a per-partition join against a partitioned table

The partitioned table implementation in SQL Server 2008 does have a quirk that is worth noting because it may surprise you at first. If you look closely at the showplan output in Figure 8-48 for this last query plan, you may notice that this partitioned table heap scan has a seek predicate.

⊞ Output List	[s1].[dbo].[testscan].randomnum, [s1].[dbo].[testscan].value, [s1].[dbo].[testscan].data
Parallel	False
Partitioned	True
Physical Operation	Table Scan
Scan Direction	FORWARD
⊞ Seek Predicates	Seek Keys[1]: Prefix: PtnId1001 = Scalar Operator([Expr1008])
TableCardinality	100000

Figure 8-48: Seek predicate for partitioned heaps

Although SQL Server 2005 exposed the partition ID within the query plan, SQL Server 2008 largely hides that from view. It is still in the query plan, but it is much more closely tied to indexing in most cases. Every partitioned access structure in SQL Server 2008 is modeled as an index where the first column is the partitioning column. Because the partitioning ID (derived from the partitioning key) is needed to perform seeks anyway, this actually matches the effective behavior seen in SQL Server 2005. The only quirk is that partitioned heaps now appear to have an index. You can see this in the properties from the previous example.

Partition-Aligned Index Views

SQL Server 2008 now allows for partition-aligned index views that can survive across *SWITCH* operations. In SQL Server 2005, these views had to be dropped before a *SWITCH* could be performed, and this hampered the ability to keep a system running as a production system while the indexed views were disabled and rebuilt. Now, partitioned tables, especially in large data warehouses, have a way to maintain a database while keeping it fully available.

Data Warehousing

SQL Server contains a number of special optimizations that speed the execution of Data Warehouse queries. A *data warehouse* is a large database that usually has one large fact table and a number of smaller dimension tables that contain detail information referenced by the fact table. These are typically called *star schema* or *snowflake schema* (*snowflake* applies to dimension tables that reference other dimension tables). These kinds of schemas are often used to store large amounts of raw data which is then processed to help discover information to help a company learn something about its business.

Data warehouses often try to make each row in the fact table as small as possible because the table is so large. Large data, such as strings, is moved to dimension tables to reduce in-row space. Fact tables are usually so large that the use of nonclustered indexes is limited because of the large storage requirements to store these structures. Dimension tables are often indexed. This pattern does not match a typical transaction processing system, where each table is accessed based on the queries used against the system.

When optimizing queries against data warehouses, it is important not to scan the fact table more than necessary because this is usually the largest single contributor to execution time. SQL Server can recognize star and snowflake schemas and apply special optimizations to improve query performance. First, SQL Server orders joins differently in data warehouses to try to perform as many limiting operations against the dimension tables as is possible *before* a scan of the fact table is performed. This can even include performing full cross products between dimension tables so that scans of the fact table can be eliminated.

SQL Server 2008 also contains improvements to bitmap operators which help reduce data movement across threads in parallel queries. The bitmap can be used to reduce each row to a single bit. Because two bitmaps can be intersected or unioned efficiently, this model allows SQL Server to join two tables simply by performing a bitmap operation. This allows each dimension table to be queried to identify qualifying rows, creating a bitmap that is then sent to the thread(s) scanning the fact table. These bitmaps are applied using a probe filter applied as a non-sargable predicate to the fact table. This is somewhat like a special on-the-fly index, created just for data warehouse queries of this pattern.

One limitation in SQL Server that still affects large tables, such as the fact table in a data warehouse configuration, is that there can only be 200 steps in a histogram. Very large tables often have more than 200 steps of interesting data distribution data, so sometimes queries may be overestimated or underestimated as a result of this limitation, even with full-scan statistics. Luckily, filtered statistics can be used to alleviate this problem somewhat—it is possible to create filtered statistics for the range that defines a partition and then have that be used to estimate cardinality. As many queries on partitioned tables are over the current partition in a date range, this covers a reasonable number of the scenarios that were not covered as well in SQL Server 2005.

Updates

Updates are an interesting area within query processing. In addition to many of the challenges faced while optimizing traditional *SELECT* queries, update optimization also considers physical optimizations such as how many indexes need to be touched for each row, whether to process the updates one index at a time or all at once, and how to avoid unnecessary locking deadlocks while processing changes as quickly as possible. The Optimizer contains a number of features that are specific to updates that help make queries complete as quickly as possible. In this section, we discuss a number of these optimizations.

In this section, the term *update processing* actually includes all top-level commands that change data. This includes *INSERT, UPDATE, DELETE,* and (as of SQL Server 2008) *MERGE*. As you see in this section, SQL Server treats these commands almost identically. Every update query in SQL Server is composed of the same basic operations:

- Determines what rows are changed (inserted, updated, deleted, merged)
- Calculates the new values for any changed columns
- Applies the change to the table and any nonclustered index structures.

Figure 8-49 shows how *INSERT* works using this pattern.

```
CREATE TABLE update1 (col1 INT PRIMARY KEY IDENTITY, col2 INT, col3 INT);
INSERT INTO update1 (col2, col3) VALUES (2, 3);
```

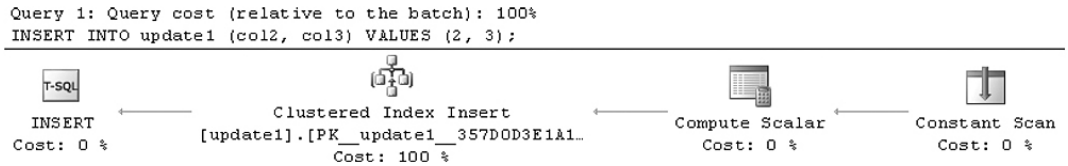


Figure 8-49: Basic INSERT query plan

The *INSERT* query has an operator called a *Constant Scan*. A Constant Scan is a special operator in the relational algebra that generates rows without reading them from a table. If you are inserting a row into a table, it doesn't really have an existing table, so this operator creates a row for the insert operator to process. The *Compute Scalar* operation evaluates the values to be inserted. In our example, these are constants, but they could be arbitrary scalar expressions or scalar subqueries. Finally, the insert operator physically updates the primary key-clustered index.

Figure 8-50 shows how *UPDATE* plans are represented.

```
UPDATE update1 SET col2 = 5;
```

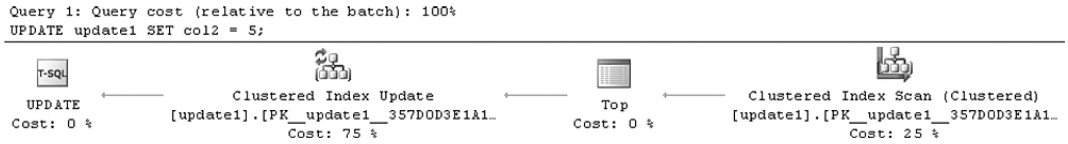


Figure 8-50: UPDATE query plan

The *UPDATE* query reads values from the clustered index, performs a *Top* operation, and then updates the same clustered index. The *Top* operation is actually a placeholder for processing *ROWCOUNT,* and it does nothing unless you have executed a *SET ROWCOUNT N* operation in your session. Also note that in the example, the *UPDATE* command does not modify the key of the clustered index, so the row in the index does not need to be moved within an index. Finally, there does not appear to be an operator to calculate the new value 5 for *col2*. This is obviously not true—it is handled, but there is a physical optimization to collapse this command into the Update operator for processing. If you examine the properties of the Update operator (as seen in Figure 8-50), you see that the query has also been auto-parameterized and the target value is supplied directly into the Update. Figure 8-51 shows the *DELETE* query plan pattern.

```
DELETE FROM update1 WHERE col3 = 10;
```

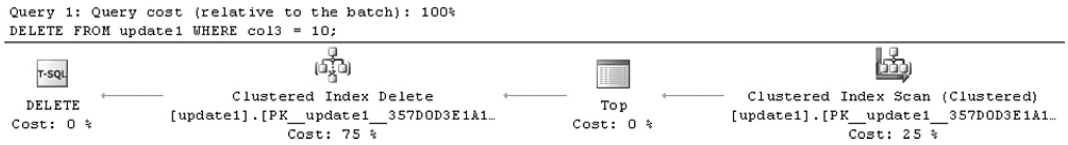


Figure 8-51: DELETE query plan

The *DELETE* query is very similar to the *UPDATE* query—the only real difference is that the row is deleted at the end. The only material difference is that the WHERE clause is used as a condition to the source table's seek operation.

SQL Server generates different plans based on the physical layout of tables, indexes, and other secondary structures. For example, if we consider a very similar example that does not have a primary key-clustered index, we can see that the resulting plan shape changes in Figure 8-52.

```
CREATE TABLE update2 (col1 INT , col2 INT, col3 INT);
INSERT INTO update2 (col2, col3) VALUES (2, 3);
```

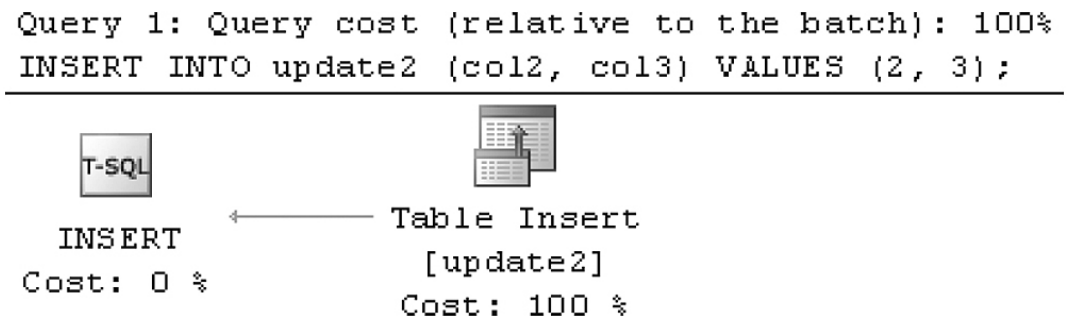


Figure 8-52: Simple INSERT query plan

When the table is a heap (it has no clustered index), a special optimization occurs that can collapse the operations into a smaller form. This is called a *simple update* (the word *update* is used generically to refer to insert, update, delete, and merge plans), and it is obviously faster. This is a single operator that does all the work to insert into a heap, but it does not support every feature in Updates.

Figure 8-53 shows how inserts work against tables with multiple indexes.

```
CREATE TABLE update3 (col1 INT , col2 INT, col3 INT);
CREATE INDEX i1 ON update3(col1);
CREATE INDEX i2 ON update3(col2);
CREATE INDEX i3 ON update3(col3);

INSERT INTO update3(col1, col2, col3) VALUES (1, 2, 3);
```

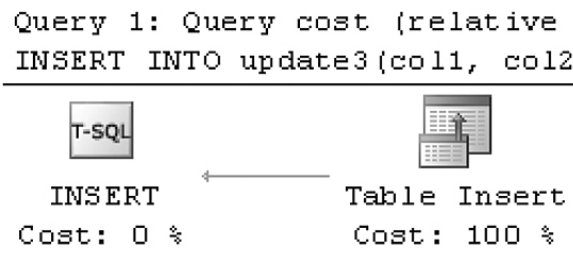


Figure 8-53: All-in-one INSERT query plan

This query needs to update all the indexes because a new row has been created. However, Figure 8-53 demonstrates that the plan has only the one operator. If you look at the properties for this operator in Management Studio, as shown in Figure 8-54, you can see that it actually updates all indexes in one operator.

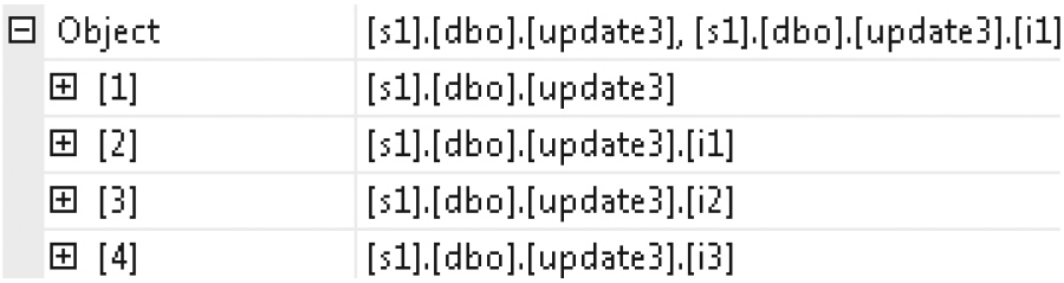


Figure 8-54: Multiple indexes updated by a single operator

This is another one of the physical optimizations that are done to improve the performance of common update scenarios. This kind of insert is called an *all-in-one* or a *per-row* insert.

Using the same table, we can try an *UPDATE* command to update some, but not all, of the indexes. Figure 8-55 contains the resulting query plan.

```
UPDATE update3 SET col2=5, col3=5;
```

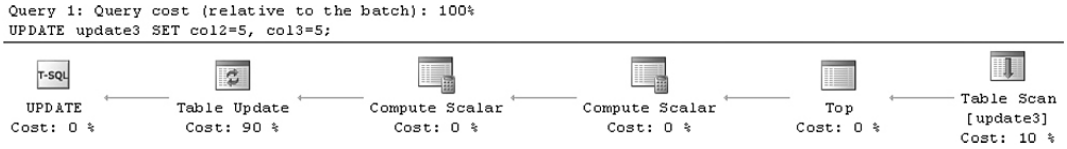


Figure 8-55: A query plan that modifies only some of the indexes on a table

Well, now things are getting a bit more complex. The query scans the heap in the Table Scan operator, performs the *ROWCOUNT Top*, then two Compute Scalars, and then a Table Update. If you examine the properties for the Table Update, you see that it lists only indexes *i2* and *i3*, because the Query Optimizer can statically determine that *i1* will not be changed by this command. One of the Compute Scalars calculates the new values for the columns. The other is yet another physical optimization that helps compute whether each row needs to modify each and every index. SQL Server contains logic to handle non-updating updates. In this case, the user calls for an update but actually submits the existing value for the row. The Query Optimizer can recognize this case and avoid some internal steps, such as logging changes, when a value is updated to the same value. Because a number of prepackaged SQL applications and tools allow users to retrieve a row, modify some columns, then write a complete update for all columns back to the database (not just the columns that changed), this actually turned out to be a needed and useful way to speed up queries. This optimization is not always applied—SQL Server uses logic to make an educated guess as to how likely and useful this optimization is, but it does reduce write traffic and log traffic in the cases when it applies.

Halloween Protection

Halloween Protection describes a feature of relational databases that is used to provide correctness in update plans. The need for the solution is best described by explaining what happens in a naive implementation of an update plan. One simple way to perform an update is to have an operator that iterates through a B+ tree index and updates each value that satisfies the filter. This works fine so long as one assigns a value to a constant or to a value that does not apply to the filter. If one is not careful, the iterator can see rows that have already been processed earlier in the scan because the

previous update moved the row ahead of the cursor iterating through the B+ tree.

Not every query needs to worry about this problem, but it is an issue for some shapes of query plans. The typical protection against this problem is to scan all the rows into a buffer, then process the rows from the buffer. In SQL Server, this is usually implemented using a Spool or a Sort operator, each of which has certain guarantees about reading all input rows before producing output rows to the next operator in the query tree. SQL Server also can use a special form of the Compute Scalar operator to provide Halloween Protection in certain limited cases, but the showplan has no public information to indicate that this is happening (other than an extra Compute Scalar being in the plan). In all cases, the copy protects against seeing the same row twice.

Split/Sort/Collapse

SQL Server contains a physical optimization called *Split/Sort/Collapse,* which is used to make wide update plans more efficient. The feature examines all the change rows to be changed in a batch and determines the net effect that these changes would have on an index. Unnecessary changes are avoided, which can reduce the I/O requirements to complete the query. This change also allows a single, linear pass to be made to apply changes to each index, which is more efficient than a series of random I/Os. Figure 8-56 contains the resulting query plan.

```
CREATE TABLE update5(col1 INT PRIMARY KEY);
INSERT INTO update5(col1) VALUES (1), (2), (3);
UPDATE update5 SET col1=col1+1;
```

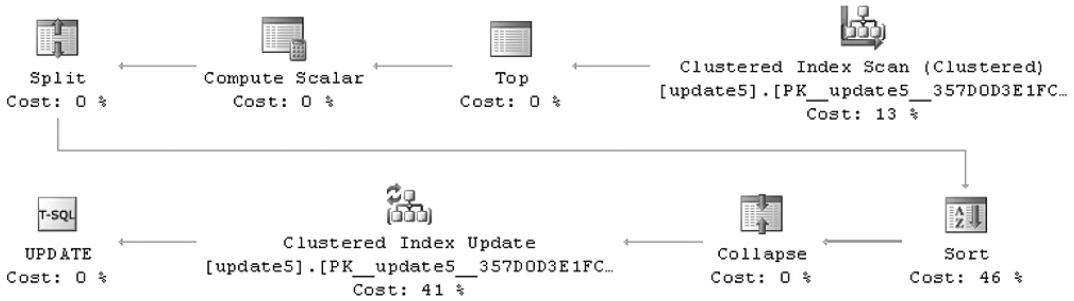


Figure 8-56: Split/Sort/Collapse UPDATE query plan

This query is modifying a clustered index that has three rows with values 1, 2, and 3. After this query, we would expect the rows to have the values 2, 3, and 4. Instead of modifying three rows, it is possible to determine that we can just delete 1 and insert 4 to make the changes to this query. For our trivial example, we can avoid the modification of one row, but for larger tables, this savings can be substantial.

This optimization is implemented using an internal column called the *action column.* It contains a value to represent whether each row is an *INSERT, UPDATE, DELETE,* or *MERGE*. The action column is used by the Update operator to determine what change should be applied to the index. Although the showplan shows different names for this Update operator based on the submitted query, it is the same operator internally and is modified by the action column. Unfortunately, you can't see the values of this column because it is only a construct within the query processor.

The action column is also used by the query processor to help determine the net changes to be applied to an index. It also is used by the Split/Sort/Collapse logic to determine the next change to the index. Let's walk through what happens in each step. Before the split, the row data is shown in Table 8-1.

Table 8-1: Pre-Split Update Data Representation

Action	Old Value	New Value
UPDATE	1	2
UPDATE	2	3
UPDATE	3	4

Split converts each *UPDATE* into one *DELETE* and one *INSERT*. Immediately after the split, the rows now appear as shown in Table 8-2.

Table 8-2: Post-

Split Data Representation

Action	Value
DELETE	1
INSERT	2
DELETE	2
INSERT	3
DELETE	3
INSERT	4

The Sort sorts on (value, action), where *DELETE* sorts before *INSERT*. After the sort, the rows appear as seen in Table 8-3.

Table 8-3: Post-Sort Data Representation

Action	Value
DELETE	1
DELETE	2
INSERT	2
DELETE	3
INSERT	3
INSERT	4

The Collapse operator looks for (*DELETE, INSERT*) pairs for the same value and removes them. In this example, it replaces the *DELETE* and *INSERT* rows with *UPDATE* for the rows with the values 2 and 3. The *UPDATE* reduces the number of B+ tree maintenance operations necessary, and the storage engine is instrumented not to log anything for B+ tree updates to the same value (locks are still taken, however, for correctness). The final form of the rows after the collapse is Table 8-4.

Table 8-4: Post-Collapse Data Representation

Action	Value
DELETE	1
UPDATE	2
UPDATE	3
INSERT	4

The result is the net change that needs to be made to the index. Technically, each index also contains a primary key reference or heap row identifier, and even the rows missing from Table 8-5 are actually updated to fix the reference to the heap or clustered index. Log traffic is still reduced from the regular update path, and the I/O ordering benefits are also gained.

While the Split/Sort/Collapse logic is a performance optimization, it also helps avoid false failures when modifying a unique index (such as this primary key). If the original plan were to be executed without Split/Sort/Collapse, it would try to change the row with value 1 to 2. This would conflict with the existing row that has value 2 in the index. Although this could be avoided for this query by iterating over the rows backwards, it is not always possible to pick a single scan order to avoid this issue. Split/Sort/Collapse allows SQL Server to support queries such as this example without returning an error.

Merge

SQL Server 2008 introduces a new type of update operation called *MERGE*. *MERGE* is a hybrid of the other update operations and can be used to perform conditional changes to a table. The business value of this operation is that it can collapse multiple T-SQL operations into a single query. This simplifies the code that you have to write to modify tables, improves performance, and really helps operations against large tables that could be so large as to make multistep operations effectively too slow to be useful.

Now that you have seen how the other update operations are handled, you might have figured out that *MERGE* is actually not a difficult extension of the action column techniques used in the other operations. Like the other queries, the source data is scanned, filtered, and modified. However, in the case of *MERGE*, the set of rows to be changed is then joined with the target source to determine what should be done with each row. Based on this join, the action column for each row is modified to tell the *STREAM UPDATE* operation what to do with each row.

In Listing 8-15, an existing table is going to be updated with new data, some of which might already exist in the table. Therefore, *MERGE* is used to determine only the set of rows that are missing. Figure 8-57 contains the resulting *MERGE* query plan.

Listing 8-15: A MERGE Example

```
CREATE TABLE AnimalsInMyYard(sightingdate DATE, Animal NVARCHAR(200));
GO
INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2008-08-12', 'Deer');
INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2008-08-12', 'Hummingbird');
INSERT INTO AnimalsInMyYard(sightingdate, Animal) VALUES ('2008-08-13', 'Gecko');
GO
CREATE TABLE NewSightings(sightingdate DATE, Animal NVARCHAR(200));
GO
INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2008-08-13', 'Gecko');
INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2008-08-13', 'Robin');
INSERT INTO NewSightings(sightingdate, Animal) VALUES ('2008-08-13', 'Dog');
GO

-- insert values we have not yet seen - do nothing otherwise
MERGE AnimalsInMyYard A USING NewSightings N
    ON (A.sightingdate = N.sightingdate AND A.Animal = N.Animal)
WHEN NOT MATCHED
    THEN INSERT (sightingdate, Animal) VALUES (sightingdate, Animal);
```

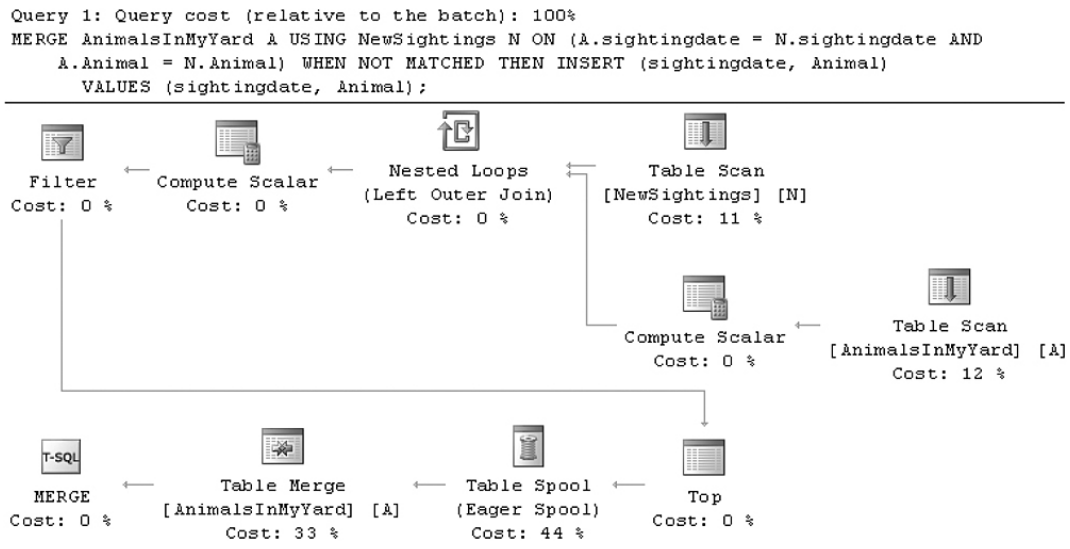


Figure 8-57: Merge query plan

As *MERGE* plans tend to get a bit large, I'll split this into pieces and discuss each portion of the query plan. The first part of the plan can be seen in Figure 8-58.

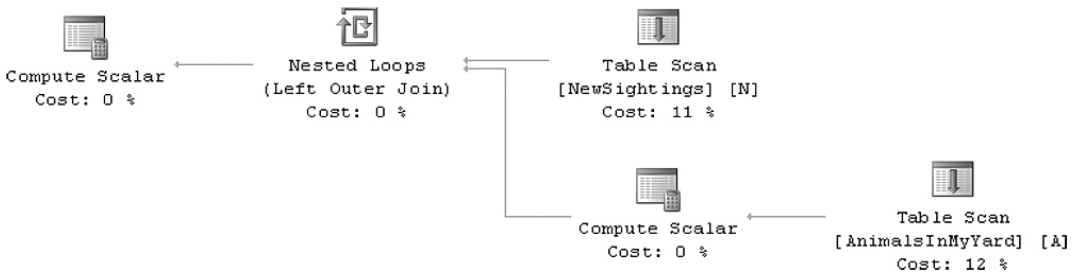


Figure 8-58: MERGE plan—initial join to find preexisting rows

First, the source table *NewSightings* is read, and the query processor performs a probe into the target table *AnimalsInMyYard* to see if the row is already there. The Compute Scalar underneath the Left Outer Join exists merely to add a column that is 1 if the value was matched and, due to the nature of how Left Outer Joins work, returns a value of NULL if there is no matching row to match the source table row. The Compute Scalar above the join generates the *Action* column:

```
[Action1008] = Scalar Operator(ForceOrder(CASE WHEN [TrgPrb1006] IS NOT NULL THEN NULL ELSE
(4) END))
```

In the upper half of this plan (shown in Figure 8-59), the filter eliminates rows that have a null action (Predicate: [Action1008] IS NOT NULL), as this *MERGE* statement only has one action (It is possible to have multiple operations within a single *MERGE* statement). The Spool provides Halloween Protection, which means that it consumes all rows from its inputs before attempting to write values back into the *AnimalsInMyYard* table. Table *MERGE* is really just an *Update* operation, but the showplan output has been changed to avoid confusion.

Figure 8-59: MERGE plan—Update, Halloween protection spool, and row filter

Wide Update Plans

SQL Server also has special optimization logic to speed the execution of large batch changes to a table. If a large percentage of a table is being changed by a query, SQL Server can create a plan that avoids modifying each B+ tree with many individual updates. Instead, it can generate a per-index plan that determines all the rows that need to be changed, sorts them into the order of the index, and then applies the changes in a single pass through the index. This approach can be noticeably more efficient than updating each row individually. These plans are called *per index* or *wide update* plans, as you see in their plan shape.

The following example demonstrates a wide update plan. Figure 8-60 contains the resulting plan.

```
CREATE TABLE dbo.update6(col1 INT PRIMARY KEY, col2 INT, col3 INT);
CREATE INDEX i1 ON update6(col2);
GO
CREATE VIEW v1 WITH SCHEMABINDING AS SELECT col1, col2 FROM dbo.update6;
GO
CREATE UNIQUE CLUSTERED INDEX i1 ON v1(col1);
UPDATE update6 SET col1=col1 + 1;
```

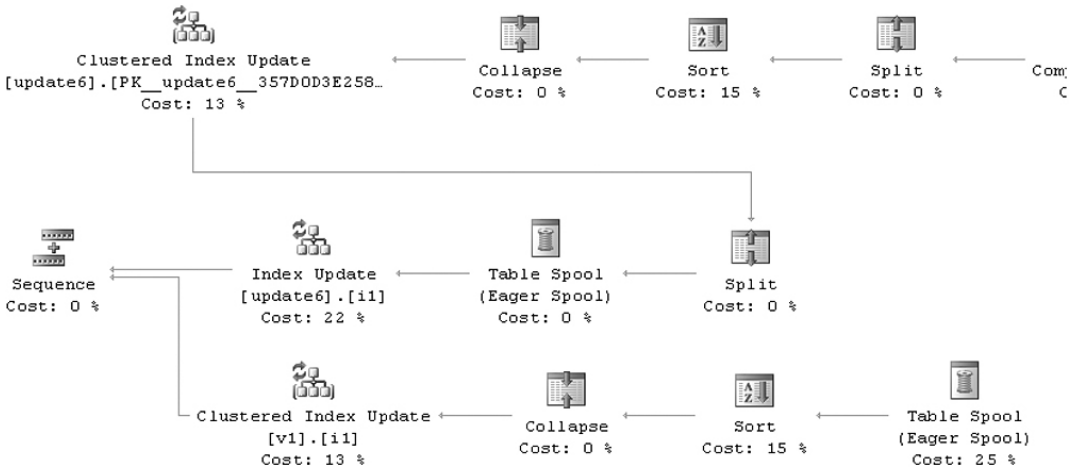


Figure 8-60: Wide update query plan (truncated)

Because this is complicated, let's split the plan into smaller sections so that it can fit on the printed page and not be as overwhelming.

Figure 8-61 contains the first portion of the query plan, and it works just like the previous example—the set of net changes are applied to the clustered index. This set is a superset of all the nonclustered indexes because a clustered index includes all columns.

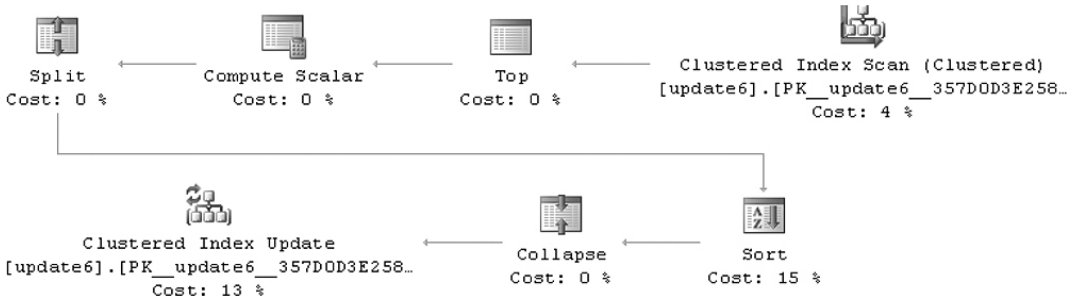


Figure 8-61: Clustered index update section of wide update plan

Figure 8-62 contains the next part of the query plan. The section of the first branch above the clustered index update does a number of things. The spool in this plan is a common subexpression spool (described earlier in the chapter). This is a way of broadcasting the rows to allow each index to use this data as input. The Sequence operator does not change or modify data—it is designed to process the first input first, the second input second, and so on. This drives the processing of the rows in a wide update plan. Finally, because there can be multiple clients of this set of rows that each perform Split/Sort/Collapse, SQL Server has an optimization to perform the split once instead of *N* times by doing it on the first branch.

Figure 8-62: Updated rows are split and stored in a multiread spool

Finally, the second branch reads the previous spooled and split rows, sorts them for this index, collapses them, and performs the net change to this index. Figure 8-63 contains this portion of the plan. If the query had additional indexes to update, they could be applied as additional branches in the query that would be processed in order.

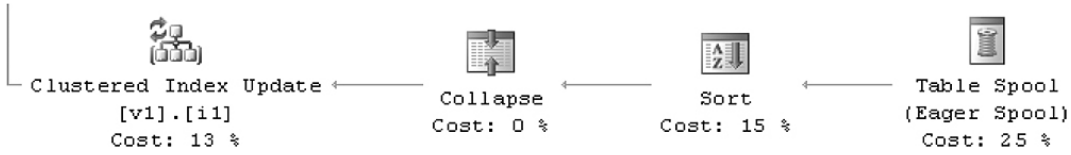


Figure 8-63: One nonclustered index branch in a wide update plan

Note Wide update plans are the most general and fully functional form of update plan in SQL Server, and architecturally any plan can be executed as a wide plan in SQL Server. Some features, such as indexed views and query notifications, are updated using only wide update plans. Because some optimizations available in SQL Server are limited to more traditional feature sets, be aware that using some features forces SQL Server to use wide update plans. In most cases, this does not matter to your application, but it could matter in systems with small amounts of data that perform many updates.

Non-Updating Updates

UPDATE operations have a number of special optimizations to improve performance for common scenarios. For example, a common programming paradigm for updating a row against a database is as follows:

1. Run a *SELECT* query that retrieves a row from the database and copies the values into the client or mid-tier layer, such as:
   ```
   SELECT col1, col2, col3, . . . FROM Table WHERE primarykey = <constant>
   ```
2. Allow the user to update some columns selectively.
3. When the client is finished modifying the row, attempt to write the values back to the server with a query like this:
   ```
   UPDATE Table SET col1=@p1, col2=@p2, col3=@p3 . . . WHERE primarykey = constant AND col
   = originalcol1value AND col2 = originalcol2value AND col3 = originalcol3value AND . . .
   ```

This pattern provides a functional but not optimal concurrency control without requiring the server to hold locks on the base table. In addition, the database programmer generally implements only one *UPDATE* query that can handle any set of modified columns and just passes in the original values in the SET list to avoid having to deal with many query plans.

SQL Server has update logic that can determine the set of indexes to maintain based on the columns in the SET list of an *UPDATE*. By default, however, the pattern described here would cause SQL Server to update all indexes for each *UPDATE*, even if only one column value actually changed. To avoid this problem, SQL Server implements a feature called *Non-Updating Updates*, which can dynamically detect unchanged values and avoid updates to unchanged indexes. While the query plan still reference each index, it is possible to avoid the cost to write unneeded values.

This optimization is not performed in all cases—some logic is used to try to apply it to scenarios where it seems most likely to improve performance. This optimization is transparent to the user, though you can see it as additional filters in some query plans.

Sparse Column Updates

SQL Server 2008 introduced a new feature called *sparse columns* that supports creating more columns in a table than were previously supported, as well as creating rows that were technically greater than the size of a database page. The primary use case for the feature is flexible-schema systems where users can create columns dynamically. Often these columns are mostly NULL but have some set of rows where a value is defined. This pattern is also largely independent for each sparse column, meaning that a given row potentially has a few but usually not many non-NULL sparse column values.

Sparse columns are stored in a complex column in a regular data row, as described in Chapter 7, “Special Storage.“ When working with the sparse column data, SQL Server must interpret the complex columns to determine which columns actually have values. To modify sparse columns, rows are read, new values are computed, and then rows are written. The main difference is that sparse columns require a bit more work to read and modify.

Partitioned Updates

Updating partitioned tables is somewhat more complicated than nonpartitioned equivalents. Instead of a single physical table (heap) or B+ tree, the query processor has to handle one heap or B+ tree *per partition.* It needs to figure out where each row belongs, and rows can move between partitions in some update plans. In addition, each index can be partitioned using a separate partition function. Even indexed views can be partitioned, and they too can be partitioned differently than the other access paths associated with a table. Luckily, partitioned update plans SQL Server 2008 are an extension of the update plan shapes already discussed in this chapter. So this section discusses only how those plans are different when using partitioning.

In the description of *SELECT* plans over partitioned tables earlier in this chapter, you may recall that the partitioning ID was represented within the query processor as a virtual leading column on every access method (heap or index). Partitioned table updates also use this representation, which makes the plans look a lot like the plans to update indexes. This leading column also appears in some of the other operators used in update plans, such as the Split/Sort/Collapse operators. The following examples demonstrate how partitioning fits into these plans.

In the first example, we create a partitioned table and then insert a single row into it. The plan can be seen in Figure 8-64.

```
CREATE PARTITION FUNCTION pfinsert(INT) AS RANGE RIGHT FOR VALUES (100, 200, 300);
CREATE PARTITION SCHEME psinsert AS PARTITION pfinsert ALL TO ([PRIMARY]);
go
CREATE TABLE testinsert(ptncol INT, col2 INT) ON psinsert(ptncol);
go
INSERT INTO testinsert(ptncol, col2) VALUES (1, 2);
```

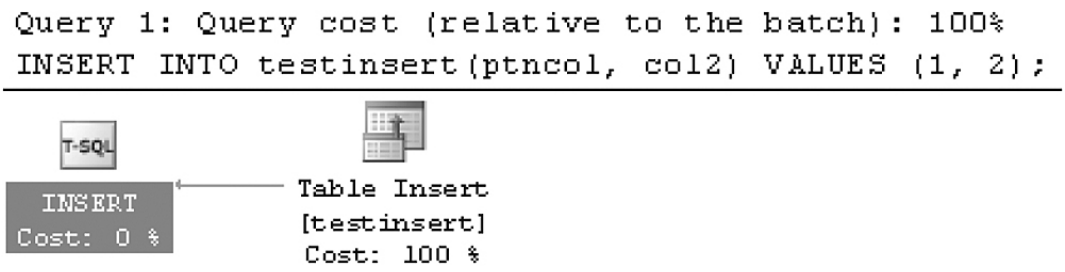
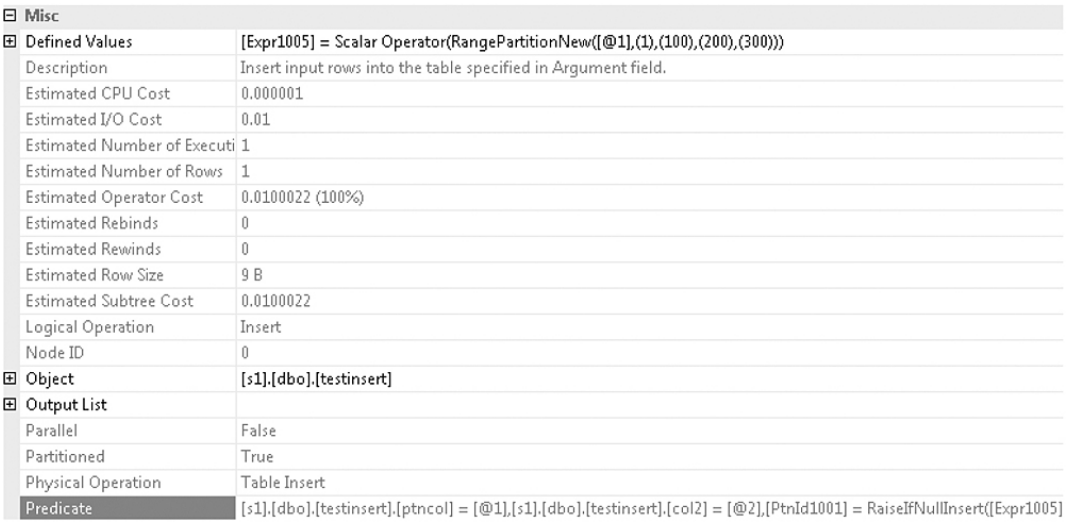


Figure 8-64: Partitioned insert—single partition

Looking at the query plan, this matches the behavior one would expect from a nonpartitioned table. It has a single operator that just inserts into the table. Underneath the covers, however, the Query Execution operator has to determine which partition needs to be updated, load that partition, and set the appropriate value. Looking at the properties for the *INSERT* operator, we can see (in Figure 8-65) the partitioning-specific logic that makes this happen. First, *Expr1005* is computed to determine the target partition to use, and you can see the partition boundaries passed to an internal function called *RangePartitionNew.* In the *Predicate* section, the extra *Expr1005* computed value is used to set the *PtnId1001* column, which is the virtual partition ID column that is exposed in the system to support partitioning. The rest of the *Predicate* list supports setting values for the regular columns *ptncol* and *col2*.



Misc	
Defined Values	[Expr1005] = Scalar Operator(RangePartitionNew([@1],(1),(100),(200),(300)))
Description	Insert input rows into the table specified in Argument field.
Estimated CPU Cost	0.000001
Estimated I/O Cost	0.01
Estimated Number of Executi	1
Estimated Number of Rows	1
Estimated Operator Cost	0.0100022 (100%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	9 B
Estimated Subtree Cost	0.0100022
Logical Operation	Insert
Node ID	0
Object	[s1].[dbo].[testinsert]
Output List	
Parallel	False
Partitioned	True
Physical Operation	Table Insert
Predicate	[s1].[dbo].[testinsert].[ptncol] = [@1],[s1].[dbo].[testinsert].[col2] = [@2],[PtnId1001] = RaiseIfNullInsert([Expr1005]

Figure 8-65: Partition selection computation in the query plan

The query processor can load the partition necessary to modify each row dynamically. If we insert multiple rows in a single statement, we can see in Figure 8-66 how the query processor supports updating each row properly.

```
INSERT INTO testinsert(ptncol, col2) VALUES (5, 10),(105,25);
```

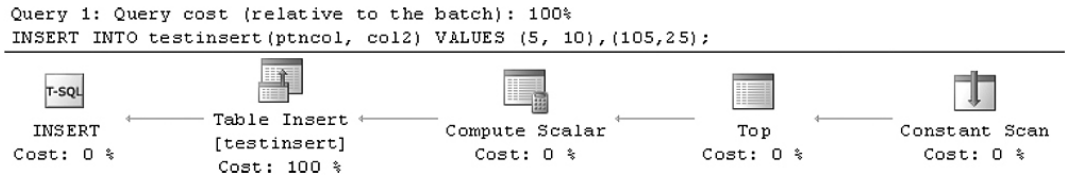


Figure 8-66: Dynamic partition computation in insert plans

This query attempts to insert two rows, and the query plan represents this using the Constant Scan operator. The Compute Scalar operator runs the partitioning function to determine the target partition of each row, as seen in Figure 8-67.

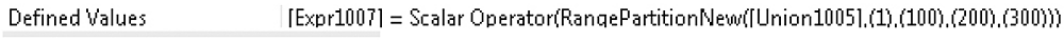


Figure 8-67: Range partitioning computation in showplan output

Furthermore, the Table Insert operator uses this computed scalar (as shown in Figure 8-68) and, for each row, changes to the right partition if necessary.

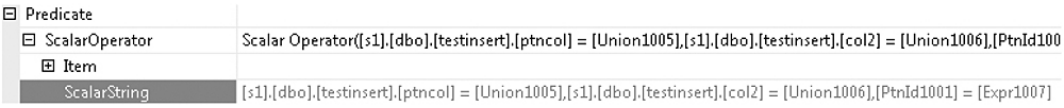


Figure 8-68: Insert uses a range partition to determine insert target partition

Note Compute Scalar exists in this two-row plan and not in the first example for reasons connected purely to implementation; these factors are not necessary to understand the plans or materially affect the performance of the plans when they are run.

Changing partitions can be a somewhat expensive operation, especially when many of the rows in the table are being changed in a single statement. The Split/Sort/Collapse logic can also be used to reduce the number of partition switches that happen, improving run-time performance. In the following example, a large number of rows are inserted into the partitioned table, and the Query Optimizer chooses to sort on the virtual partition ID column before inserting to reduce the number of partition switches at run time. Figure 8-69 shows the Sort optimization in the query plan.

```
INSERT INTO testinsert SELECT * FROM #nonptn;
```

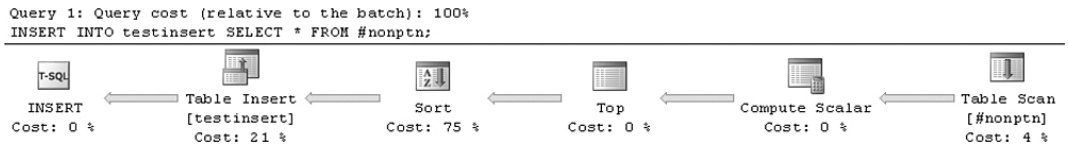


Figure 8-69: Sort optimization to reduce partition switching

The sort has an ordering requirement, as defined in Figure 8-70, which is derived from the call to the partitioning function in the Compute Scalar earlier in the plan, just like the previous example.

Figure 8-70: Ordering requirement for partitioned insert sort optimization

Updates to partitioned tables are more complex because they can move rows. However, they follow the same principles as updates. The key property to understand is that the query processor must read each row, determine the change to that row, compute the target partition for that row, and then perform the change. This may include deleting the partition from one B+ tree and inserting it into another. This matches closely with the Split/Sort/Collapse idea for batch updates, but for partitioned updates, it can happen even for smaller changes.

Locking

SQL Server contains a number of tricks and optimizations to improve the overall performance and throughput of updates

within the system. While much of the Query Optimizer is agnostic to locking, several targeted features and locking modes in Updates improve concurrency (and avoid deadlock errors). One special locking mode is called a U (for Update) lock. This is a special lock type that is compatible with other S (shared) locks but incompatible with other U locks. In many of the plan shapes used in *Update* queries, SQL Server has two different operators accessing the same access method. The first one is the source table, and it is only reading. The second is the update itself. If only a shared (S) lock were taken in the read operator, multiple users could run queries at the same time, both acquire S locks for a row and then deadlock because neither could upgrade the lock to an exclusive (X) lock when the update operator later saw the row. To prevent this, the U lock is compatible with other S locks but not with other U locks. This prevents other potential writers from reading a row, which then avoids the deadlock.

Listing 8-16 demonstrates how to examine the locking behavior of an update query plan. Figure 8-71 contains the query plan used in this example, and Figure 8-72 contains the locking output from *sp_lock*.

Listing 8-16: Update Locking Example

```
CREATE TABLE lock(col1 INT, col2 INT);
CREATE INDEX i2 ON lock(col2);
INSERT INTO lock (col1, col2) VALUES (1, 2);
INSERT INTO lock (col1, col2) VALUES (10, 3);

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
UPDATE lock SET col1 = 5 WHERE col1 > 5;
EXEC sp_lock;
ROLLBACK;
```

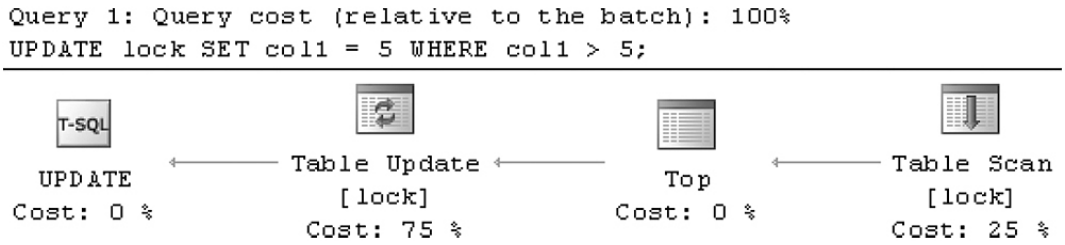


Figure 8-71: An update plan used in the locking example

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	52	21	0	0	DB		S	GRANT
2	52	1	1131151075	0	TAB		IS	GRANT
3	52	21	693577509	0	PAG	1:75676	IX	GRANT
4	52	21	693577509	0	RID	1:75676:1	X	GRANT
5	52	21	693577509	0	TAB		IX	GRANT

Figure 8-72: The sp_lock output for the update plan

Using a higher isolation mode with a user-controlled (non-auto-commit) transaction allows you to examine the final locking state of each object in the query. In this case, you can see that the row (*Resource 1:69641:1*) was locked with an X lock. This lock started as a U lock and was promoted to an X lock by the *UPDATE*.

We can run a slightly different query that shows that the locks vary based on the query plan selected. Figure 8-73 contains a seek-based update plan. In the second example, the U lock is taken by the nonclustered index, whereas the base table contains the X lock. So, this U lock protection works only when going through the same access paths because it is taken on the first access path in the query plan. Figure 8-74 shows the locking behavior of this query.

```
BEGIN TRANSACTION;
UPDATE lock SET col1 = 5 WHERE col2 > 2;
EXEC sp_lock;
```

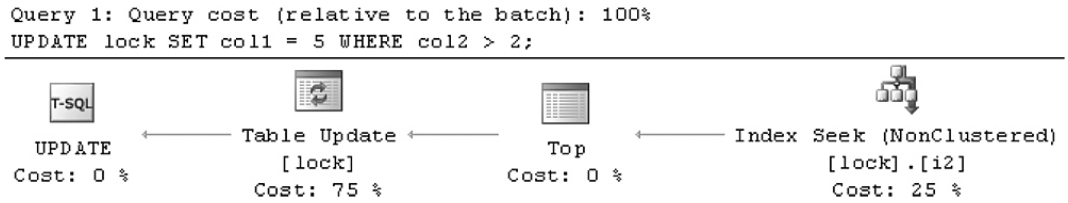


Figure 8-73: Locking behavior of an update plan with a seek

	spid	dbid	ObjId	IndId	Type	Resource	Mode	Status
1	52	21	0	0	DB		S	GRANT
2	52	21	693577509	2	KEY	(a0004dc87aeb)	U	GRANT
3	52	1	1131151075	0	TAB		IS	GRANT
4	52	21	693577509	2	PAG	1:75678	IU	GRANT
5	52	21	693577509	0	PAG	1:75676	IX	GRANT
6	52	21	693577509	0	RID	1:75676:1	X	GRANT
7	52	21	693577509	0	TAB		IX	GRANT

Figure 8-74: The sp_lock output for a seek-based update plan

Partition-Level Lock Escalation

Locking behavior is usually not the domain of the Query Optimizer. Although the Query Optimizer does try to generate plans that minimize locking conflicts, it is largely agnostic to the locking interactions of plans, for better or worse. The Query Optimizer uses a lot of logic to implement partitioning, including logic to prune unnecessary partitions from query plans so that they are not touched. One great addition in the SQL Server 2008 product is *partition-level lock escalation*. This feature allows the database to avoid lock escalation to the table-level for partitioned tables. When combined with pruning, this provides a powerful way to improve application concurrency, especially when queries over large, partitioned tables can take a long time to execute. The functionality can be enabled using the following command:

```
ALTER TABLE TableName SET (LOCK_ESCALATION = AUTO);
```

Locking is discussed in detail in Chapter 10, “Transactions and Concurrency.“

Distributed Query

SQL Server includes a feature called *Distributed Query*, which accesses data on different SQL Server instances, other database source, and non-database tabular data such as Microsoft Office Excel files or comma-separated text files. Distributed Query is based on the OLE DB interfaces, and most OLE DB providers are feature-rich enough to be used by the Distributed Query feature. Because multiple sources can be referenced within a single query, it is an effective mechanism to interact with data from multiple sources without writing a lot of special-case code.

Distributed Query supports several distinct use cases. First, Distributed Query is useful to move data from one source to another. Although it is not a complete extract, transform, and load (ETL) tool like SQL Server Integration Services, it is often a very easy way to copy a table from one server instance to another. For example, if a company’s financial reporting group wanted a copy of the sales figures for each month, it would be possible to write a query to copy the data from the SQL Server instance servicing the Sales team to another instance in the Financial Reporting group. Another application of Distributed Query is to integrate nontraditional sources into a SQL Server query. As there are OLE DB providers for non-database data such as Active Directory Domain Services, Microsoft Exchange Server, and a number of third-party sources, it is possible to write queries to gather information from those sources and to then use the power of the SQL language to ask rich questions of that data that may not be supported by the source of that data. Distributed Query can also be used for reporting. Because multiple sources can be queried in a single query, you can use this to gather data into a single source and generate reports (which can be surfaced through Reporting Services, if desired). Finally, Distributed Query can be used for scale-out scenarios. SQL Server supports a special *UNION ALL* view called a *Distributed Partitioned View (DPV)*. This view stitches together distinct portions of a single range that are each stored on a different SQL Server instance. Exceptionally large tables can be stored on different servers and queries can be directed to access only the subset necessary to satisfy a particular query. The Distributed Query feature covers a number of scenarios and can help make solving those scenarios much easier.

Distributed Query is implemented within the Query Optimizer's plan-searching framework. With the exception of pass-through queries, which are not modified during optimization, distributed queries initially are represented using the same operators as regular queries. Each base table represented in the Query Optimizer tree contains metadata collected from the remote source using OLE DB metadata interfaces such as OLE DB schema rowsets. The information collected is very similar to the information that the query processor collects for local tables, including column information, index information, and statistics. One additional piece of collected information includes information about what SQL grammar constructs the remote source supports, which are used later in optimization. Once metadata is collected, the Query Optimizer derives special property information for each operator that manipulates remote data. This property determines whether it is possible to generate a SQL statement to represent the whole query sub-tree that can be sent directly to the remote data source. Some operators can be remoted easily, like Filter and Project. Others can be performed only locally, such as the streaming table-valued function operator used to implement portions of the XQuery feature in SQL Server. SQL Server performs exploration rules to transform query trees into forms that might allow the server to remote larger trees. For example, SQL Server attempts to group all remote tables from the same source together in a single sub-tree and splitting aggregates into local forms that can be remoted. During this process, some of the more advanced rules in SQL Server are disabled if they are known to generate alternatives that prevent sub-tree remoting. While SQL Server does not maintain specific costing models for each remote source, the feature is designed to remote large sub-trees to that source in the hopes of moving the least amount of data between servers. This usually provides a close-to-optimal query plan.

In this example, we create a linked server to point to a remote SQL Server instance. Then, we use the four-part name syntax to generate a query that can be completely remoted to the remote source (shown in Figure 8-75).

```
EXEC sp_addlinkedserver 'remote', N'SQL Server';
go
SELECT * FROM remote.Northwind.dbo.customers WHERE ContactName = 'Marie Bertrand';
```

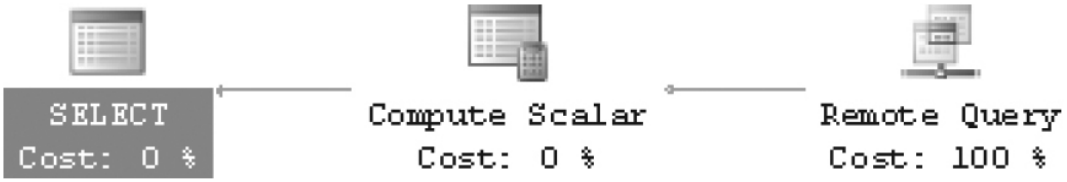


Figure 8-75: A fully remoted Distributed Query

As you can see, this relatively simple query was essentially completely remoted by the Query Optimizer. The properties information for the Remote Query node contains the query text that is executed on the remote server. The results are brought back to the local server and returned to the user.

The generated query, shown here, is somewhat more verbose than the text submitted originally, but this is necessary to ensure that the semantics of the remoted query match the local query text exactly:

```
SELECT "Tbl1002"."CustomerID" "Col1004","Tbl1002"."CompanyName" "Col1005",
"Tbl1002"."ContactName" "Col1006","Tbl1002"."ContactTitle" "Col1007","Tbl1002"."Address"
"Col1008","Tbl1002"."City" "Col1009","Tbl1002"."Region" "Col1010","Tbl1002".
"PostalCode" "Col1011","Tbl1002"."Country" "Col1012","Tbl1002"."Phone"
"Col1013","Tbl1002"."Fax" "Col1014" FROM "Northwind"."dbo"."customers" "Tbl1002" WHERE
"Tbl1002"."ContactName"=N'Marie Bertrand';
```

Because the OLE DB model has a rich, cursor-based update model, it is possible to use SQL Server to update remote data sources using regular *UPDATE* statements. These plans look identical to the local plans discussed in this chapter, except that the top-level Update operation is specific to the remote source. Because the storage engine model in SQL Server is originally based on the OLE DB interfaces, the mechanisms for performing local and remote updates are actually very similar. In the case when a whole update query can be remoted (because the property information for every operator determines that the remote source can support an *UPDATE* statement that is semantically equivalent to performing the operation locally through the exposed OLE DB interfaces), SQL Server can and will generate complete remote *INSERT, UPDATE,* and *DELETE* statements to be performed on the remote server. You should examine any query that you think can be completely remoted to make sure that it actually can—in some cases there may be a specific grammar construct or intrinsic function that is not necessary in the query that blocks it from being completely remoted.

The Distributed Query feature was introduced in SQL Server 7.0, and there are some limitations with the feature that you should consider when designing scenarios that use it. First, the feature relies on the remote providers to supply very detailed cardinality and statistical information to SQL Server so that it can use this knowledge to compare different query plans. As most OLE DB providers do not provide much statistical information, this can limit the quality of query plans

generated by the Query Optimizer. In addition, OLE DB is not being actively extended by Microsoft, and therefore some providers are not actively maintained. Also, not every feature in SQL Server is supported via the remote query mechanism, such as some XML and UDT-based functionality. SQL Server 2008 does not have a native mechanism to support querying managed adaptors written for the CLR run time. Finally, the costing model used within SQL Server is good for general use but sometimes generates a plan that is substantially slower than optimal. Unfortunately, the impact of not remoting a query in the Distributed Query feature is larger than in the local case because there is just more work to be done to move rows from a remote source. Care should be taken when using the feature to test out the functionality before you put it into production. It might be useful to pregenerate pass-through queries for common, expensive queries to make sure that they are always remoted properly.

Extended Indexes

SQL Server contains a number of special indexes to support specific use cases. Full-text indexes support document storage, querying, and retrieval. XML indexes are used to support XQuery operations on XML data stored in the database. Spatial indexes, added in SQL Server 2008, support queries over spatial data. These indexes are usually better suited for their specific domain than a B+ tree index, but they are often specific to a particular set of use cases. While the Query Optimizer contains support for each of these, they are not conceptually different from B+ tree indexes.

Full-Text Indexes

In SQL Server 2008, full-text indexes have moved from being a completely external construct to being a mostly internal index type. Specific keywords in SQL grammar tell the system to pick the index implicitly, and this is done before the Optimization process begins. Information about full-text-index generations and other details of the index is abstracted from the Query Optimizer to simplify the maintenance of this index and to avoid recompiles to account for each new index generation.

XML Indexes

XML indexes are somewhat unlike other indexes in that it is actually stored more similarly to an indexed view. It uses the same physical storage as a clustered index, and it also has secondary indexes on various columns. However, this is not matched by the Query Optimizer. Like full-text indexes, XQuery constructs are very specific in the syntax, and they are taken to imply that the operation should always use this index if it exists.

An XML index essentially takes each attribute and value in an XML document and shreds it into its own row. In addition to these values, other columns are added to this structure to store information such as the attribute for a value and its relative location in the document (its path from the root node). This information makes specific XQuery constructs significantly faster when the index is used.

Within the query tree representation, XML is represented using a number of nontraditional operators. One of these is the Streaming Table-Valued Function (STVF), which is used for other functionality, including SQL Server's Dynamic Management Objects. This construct allows an XML BLOB to be split into pieces, each returned in one row.

Spatial Indexes

Spatial indexes are new in SQL Server 2008, and these indexes are matched in the Query Optimizer. Spatial indexes decompose a space and allow points to be indexed. The primary model of a spatial index is to divide areas of the space into regions and then use bounding boxes for each region. Within the plan, an STVF generates candidates, based on the encoding function, close to the requested point(s) and the results are used in the rest of the query. Figure 8-76 contains an example spatial index query plan.

```
CREATE TABLE geo(col1 INT PRIMARY KEY, point GEOGRAPHY);
CREATE SPATIAL INDEX spaidx ON geo(point);
SELECT * FROM geo WITH (INDEX=spaidx)
    WHERE geo.point.STEquals('POINT(24.0 24.0)')=1;
```

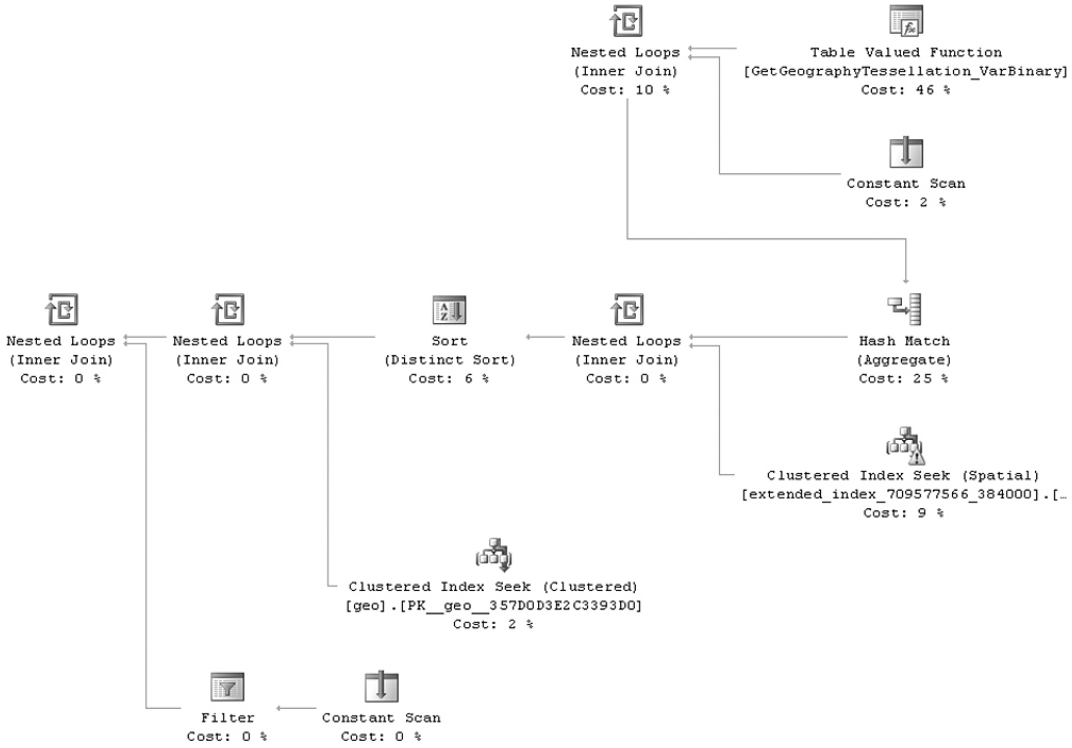


Figure 8-76: Spatial index example

The spatial index uses more operators in the plan than just the STVF. The first Constant Scan and Hash Match help reduce duplicates due to the hierarchy encoding. The first loops join retrieves values from the index structure. The sort is needed to find duplicate base rows (perhaps there is more than one point for a single database row). The second loops join retrieves any other requested columns from the base table. Finally, the last loops join applies the CLR function as a residual predicate.

Plan Hinting

There is a lot of misinformation about query hints, and often the misuse of hints can cause the creation of global policies for or against the use of hints in queries. One of the goals of this chapter is to give DBAs and developers the tools they need to have a different kind of conversation about the design, implementation, and maintenance of SQL Server applications. This section explains query hints and when to use them.

Early in the chapter, I discussed some of the problems that face the engineering team for a Query Optimizer. The complexity of some of the algorithms involved in optimizing SQL queries is high enough that it is effectively impossible to explore every possible query plan for every query that can be sent to the system. Latency restrictions in statistics gathering and mathematical modeling issues in cardinality estimation also place some limits on the powers of the Query Optimizer, given the current computational powers of processors today. The reality is that there are some queries for which the Query Optimizer cannot generate a perfect plan.

That being said, the Query Optimizer actually does an amazing job on most queries. Many years of very smart thinking has gone into the development of this component, and the result is a system that almost always finds a very good query plan very quickly. This is accomplished through a number of smart algorithms, heuristics, and an understanding of common scenarios. Each release of the product gets better and can handle more and more scenarios.

When people ask me about hints, I first ask them about their application. Many people don’t realize that the design of the application has a huge impact on whether hints are appropriate or necessary. If your database schema has a classic, third-normal form set of data tables and your queries are all written using an understanding of the American National Standards Institute (ANSI) SQL grammar, then your odds are very good that SQL Server will do a reasonable job on your query without any modification. As you push the system and stress the design in different directions, you can find areas where the algorithms and heuristics in the product start to not work as well. For example, if you have huge variations in data distribution or you have an application that relies on the statistical correlation of column values to select a great plan, then sometimes you may not get a join order that is near optimal for your query. So before you consider hints, make sure that you can understand how your application is designed, specifically around what kinds of things make your application not look like a common database application.

If you have identified a poorly performing query that is important to your application and you have an idea why the Query Optimizer could be having trouble, then that would be an appropriate time to consider whether a hint will help this application. I usually tell people not to use a hint unless there is a good reason, which means that the standard behavior of the system is unacceptable for your business and there is a better plan choice that is acceptable. So, if you know that a particular join order or index selection yields deadlocks with the other queries in your system, then you should consider using a hint—locking is not a factor in how the Query Optimizer optimizes queries, and each query is effectively optimized independently of the others in the system.

Now, some database development teams may impose rules such as "No hints," or "Always force this index on this table when doing a *SELECT*." This doesn't mean that they are wrong— often there is a very good reason for these kinds of rules to exist. I urge you to read through this chapter and make sure that you have a conversation with your DBA about the reasons for each rule. When you are building a new feature and changing a database application, it may be completely appropriate to use a hint or to alter these development practices. The goal of this section is to help you understand the purpose of each hint. I hope that this lets you see the situations when it might be appropriate to use a hint (and when it might not be appropriate).

Query and table hints are, in almost all cases, actually requirements given by the query author to the Query Optimizer when generating a query plan. So, if a hint cannot be satisfied, the Query Optimizer actually returns an error and does not return a plan at all. Locking hints are an exception—these are sometimes ignored to preserve the correctness of data manipulation operations necessary for the system to work properly. The name is somewhat misleading, but this behavior allows you to modify the query and know that you had an impact on the query generation process.

Debugging Plan Issues

Determining when to use a hint requires an understanding of the workings of the Query Optimizer, how it might not be making a correct decision, and then an understanding of how the hint might change the plan generation process to address the problem. For more than half of the problems that Microsoft typically sees in support calls about query plans, issues with incorrect cardinality estimates are the primary cause of a poor plan choice. In other words, a better cardinality estimate would yield a plan choice that is acceptable. In other cases, more complex issues around costing, physical data layout, lock escalation, memory contention, or other issues were factors in the performance degradation. This section explains how to identify cardinality estimation errors and then use hints to correct poor plan choices, as these are usually something that can be fixed without buying new hardware or otherwise altering the host machine.

The primary tool to identify cardinality estimation errors is the statistics profile output in SQL Server. When enabled, this mode generates the *actual* cardinalities for each query operator in the plan. These can be compared against the Query Optimizer's estimated cardinalities to find any differences. As cardinality estimation is performed from the bottom of the tree upwards (right to left in the showplan's graphical display), errors propagate. Usually, the location of the lowest error indicates where to consider hints.

Other tools exist to track down performance issues with query plans. Setting statistics time on is a great way to determine run-time information for a query. SQL Profiler is a great tool for tracking deadlocks and other system-wide issues that can be captured by tracing. *DBCC MEMORYSTATUS* is an excellent tool to find out what components in the system are causing memory pressure within SQL Server. Most of these tools fall outside of the scope of this chapter, though these tools can be helpful for some plan issues. It is recommended that cardinality issues be researched first when there are concerns about plan quality, as this is probably the most common issue.

In Figure 8-77, we run a query against one of the catalog views to see how each operator's estimated and actual cardinalities match up.

```
SET STATISTICS PROFILE ON;
SELECT * FROM sys.objects;
```

	Rows	Executes	StmtText	EstimateRows	EstimateExecutions
1	91	1	SELECT * FROM sys.objects;	91	NULL
2	91	1	|--Nested Loops(Left Outer Join, OUTER REFEREN...	91	1
3	91	1	|--Nested Loops(Left Outer Join, OUTER REFER...	91	1
4	91	1	| |--Filter(WHERE:(has_access('CO',[s1].[sys].[sy...	91	1
5	0	0	| | |--Compute Scalar(DEFINE:([Expr1006]=CO...	91	1
6	91	1	| | |--Clustered Index Scan(OBJECT:([s1].[sy...	91	1
7	0	91	| |--Clustered Index Seek(OBJECT:([s1].[sys].[sys...	1	91
8	91	91	|--Clustered Index Seek(OBJECT:([mssqlsystemres...	1	91

Figure 8-77: Statistics profile ouput

Note The *EstimateRows* and *EstimateExecutions* columns have been moved from the actual output order for display in the screenshot. While the estimation for this query is actually perfect, it is common for estimates to vary from the actual cardinalities, especially as queries get more complex. Usually, you'd like them to be close enough that the plan choice won't change, which is almost always less than an order of magnitude off by the top of the tree. Also, note that the *EstimateRows* number is the average per-execution, whereas *Rows* is merely total rows. You can divide *Rows* by *Executes* to get the numbers to be comparable.

If an error is found by looking at the statistics profile output, this can help identify a place where the Query Optimizer has used bad information to make a decision. Usually, updating statistics with fullscan can help isolate whether this is an issue with out-of-date or undersampled statistics. If the Query Optimizer makes a poor decision even with up-to-date statistics, then this might mean that there is an out-of-model condition with the Query Optimizer. For example, if there is a strong data correlation between two columns in a query, this could cause errors in the cardinalities seen in the query. Once an out-of-model condition is identified as being the cause of a poor plan choice, hints are the mechanism to correct the plan choice and to use a better query plan.

This section describes how most of the query or table hints fit within the context of the Query Optimizer's architecture, including situations where it might be appropriate to use hints.

{HASH | ORDER} GROUP

SQL Server has two possible implementations for *GROUP BY* (and *DISTINCT*). It can be implemented by sorting the rows and then using the fact that rows in the same group are now adjacent physically. It can also hash each group into a different memory location. When one of these options is specified, it is implemented by turning off the implementation rule for the other physical operator. Note that this applies to all *GROUP BY* operations within a query, including those from views included in the query.

Many data warehouse queries have a common pattern of a number of joins followed by an aggregate operation. If the estimates for the number of rows returned in the joins section is in error, the estimated size for the aggregate operation can be substantially incorrect. If it is underestimated, then a sort and a stream aggregate may be chosen. As memory is allocated to each operator based on the estimated cardinality estimates, an underestimation could cause the sort to spill to disk. In a case like this, hinting a hash algorithm might be a good option. Similarly, if memory is scarce or there are more distinct grouping values than expected, then perhaps using a stream aggregate would be more appropriate. This hint is a good way to affect system performance, especially in larger queries and in situations when many queries are being run at once on a system.

{MERGE | HASH | CONCAT } UNION

Many people incorrectly use *UNION* in queries when they likely want to use *UNION ALL*, perhaps because it is shorter. *UNION ALL* is actually a faster operation, in general, because it takes rows from each input and simply returns them all. *UNION* actually has to compare rows from each side and make sure that no duplicates are returned. Essentially, *UNION* performs a *UNION ALL* and then a *GROUP BY* operation over all output columns. In some cases, the Query Optimizer can determine that the output columns contain a key that is unique over both inputs and can convert the *UNION* to a *UNION ALL*, but in general, it is worth making sure that you are actually asking the right query. These three hints apply only to *UNION*.

Now, assuming that you have the right operation, you can pick among three join patterns, and these hints let you specify which one to use. This example shows the MERGE UNION hint.

```
CREATE TABLE t1 (col1 INT);
```

```
CREATE TABLE t2 (col1 INT);
go
INSERT INTO t1(col1) VALUES (1), (2);
INSERT INTO t2(col1) VALUES (1);

SELECT * FROM t1
UNION
SELECT * FROM t2
OPTION (MERGE UNION);
```

As you can see, each hint forces a different query plan pattern. MERGE UNION is useful when there are common input sizes. CONCAT UNION is best at low-cardinality plans (one sort). HASH UNION works best when there is a small input that can be used to make a hash table against which the other inputs can be compared.

UNION hinting is done for roughly the same reasons as *GROUP BY* hinting—both operations are commonly used near the top of a query definition, and they have the potential to suffer if there is error in cardinality estimation in a query with many joins. Typically, one either hints to the HASH operator to address cardinality underestimation or hints to the CONCAT operator to address overestimation.

FORCE ORDER, {LOOP | MERGE | HASH } JOIN

Join order and algorithm hints are common techniques to fix poor plan choices. When estimating the number of rows that qualify a join, the best algorithm depends on factors such as the cardinality of the inputs, the histograms over those inputs (which are used to make estimates about how many rows qualify the join condition), the available memory to store data in memory such as hash tables, and what indexes are available (which can speed up loops join scenarios). If the cardinality or histograms are not representative of the input, then a poor join order or algorithm can result. In addition, there can be correlations in data across joins that are extremely difficult to model with current technologies (even filtered statistics in SQL Server 2008 work only within a single table).

> **Tip** If the statistics profile output demonstrates that the cardinality estimates were substantially incorrect, the join order can be forced by rewriting the query into the order you would like to see the tables in the output plan. This modifies how the Query Optimizer sets the initial join order and then disables rules that reorder joins. Once hinted, you should time the new query and make sure that the plan is faster than the original. In addition, as your data changes, you need to reexamine these hints regularly to make sure that the plan you have forced is still appropriate—you are essentially saying “I know better than the Query Optimizer,” which is the equivalent of performing all the maintenance on your own car.

Places where I have seen these hints be appropriate in the past are the following:

- Small, OLTP-like queries where locking is a concern.
- Larger data-warehouse systems with many joins, complex data correlations, and enough of a fixed query pattern that you can reason about the join order in ways that make sense for all queries. For example: “I am happy if I access dimension tables in this order first, then the fact table and everything is a hash join.”
- Systems that extend beyond traditional relational application design and using some engine feature enough to change query performance materially. Examples might include using SQL as a document store with Full-Text or XQuery components that are mixed with traditional relational components or using Distributed Queries against a remote provider that does not surface statistical information to SQL Server’s query processor.

Unfortunately, no semi-join specific implementation hints are exposed in SQL Server 2008, although they can be indirectly affected by the other join hints.

INDEX=<indexname> | <indexid>

The INDEX=<indexname> | <indexid> hint has been in the product for many releases and is very effective in forcing the Query Optimizer to use a specific index when compiling a plan. The primary scenario where this is interesting is an OLTP application where you wish to force a plan to avoid scans of any type. Remember that the Query Optimizer tries to generate a plan using only this index first, but it also adds joins to additional indexes for a table if the index you have forced is not covering for the query (meaning that it has all the columns contained in the index key, the table’s primary key, or listed as an INCLUDED column). One would generally have a query filter predicate that could be used to generate a seek against the index, but this hint is also valid for index scans if you use indexes to narrow row widths to improve query execution time. A second scenario where this would be useful is a plan developed on one server for use on another, such

as a test to deployment server or an ISV that creates a plan for an application and then ships this application to their customers to deploy on their own SQL Server instance.

FORCESEEK

This hint was added in SQL Server 2008. It tells the Query Optimizer that it needs to generate a seek predicate when using an index. There are a few cases when the Query Optimizer can determine that a scan of an index is better than a seek when compiling the query. For example, if a query is compiled while the table is almost empty, the storage engine may store all the existing rows in one page in an index. In this case, a scan is faster than a seek, in terms of I/O, because the storage engine supports scanning from the leaf nodes of a B+ tree, which would avoid one extra page of I/O. This condition might be ephemeral, given that newly created tables are often populated soon after. The hint is effective in avoiding such a scenario if you know that perhaps the table won't have enough rows to trigger a recompile or that the performance impact of this condition would be detrimental enough to the system to warrant the hint.

The other scenario where such a hint is interesting is to avoid locks in OLTP applications. This hint precludes an index scan, so it can be effective if you have a high-scale OLTP application where locking is a concern in scaling and concurrency. The hint avoids the possibility of the plan taking more locks than desired. Because the Query Optimizer does not explicitly reason about locking in plan selection (it does not prefer a plan that has fewer locks, but it might prefer a plan that it things will perform faster that also happens to take fewer locks). Care should be taken in hinting high-scale applications only when necessary, as a poor hint can cause the system to behave substantially worse than an unhinted plan.

FAST <number_rows>

The Query Optimizer assumes that the user will read every row produced by a query. Although this is often true, some user scenarios, such as manually paging through results, do not follow this pattern—in these cases, the client reads some small number of rows and then closes the query result. Often a similar query is submitted in the near future to retrieve another batch of rows from the server. In the costing component, this assumption affects the plan choice. For example, hash joins are more efficient for larger result sets but have a higher startup cost (to build a hash table for one side of a join). Nested loops joins have no startup cost but a somewhat higher per-row cost. So, when a client wants only a few rows but does not specify a query that returns only a few rows, the latency of the first row may be slower due to the startup costs for stop-and-go operators like hash joins, spools, and sorts.

The FAST <number_rows> hint supplies the costing infrastructure with a hint from the user about how many rows the user will want to read from a query. Internally, this is called a *row goal*, and simply provides an input into the costing formulas to help specify what point on the costing function is appropriate for the user's query.

The *TOP()* syntax in SQL Server introduces a row goal as well. Note that if you supply *TOP(@param)*, then the Query Optimizer may not have a good value to sniff from the T-SQL context. In this scenario, you would want to use the OPTIMIZE FOR hint (described later in this section).

MAXDOP <N>

MAXDOP stands for maximum degree of parallelism, which describes the preferred degree of fan-out to be used when this query is run (in SKUs that support parallel query plans). For expensive queries, the Query Optimizer attempts to use multiple threads to reduce the run time of a query. Within the costing functions, this means that some portions of the costs of a query are divided over multiple processor cores, reducing the overall cost compared to an otherwise identical serial plan. Very complex queries can actually have multiple zones of parallelism, meaning that each zone can have up to MAXDOP threads assigned to it during execution.

Large queries can consume a nontrivial fraction of the resources available to the system. A parallel query can consume memory and threads, blocking other queries that want to begin execution. In some cases, it is beneficial to the overall health of the system to reduce the degree of parallelism for one or more queries to lower the resources required to run a long-running query. This helps workloads that do not use the resource governor to manage resources. Often a server that services mixed workloads would be a good candidate for considering this hint, when needed.

OPTIMIZE FOR

The Query Optimizer uses scalar values within the query text to help estimate the cardinality for each operator in the query. This ultimately helps choose the lowest-cost plan, as cardinality is a major input into the costing functions. Parameterized queries can make this process more difficult because parameters can change from one execution to the next. Given that

SQL Server automatically parameterizes queries as well, this design choice affects more queries than one would expect. When estimating cardinality for parameterized queries, the Query Optimizer usually uses a less accurate estimate of the average number of distinct values in the column or it sniffs the parameter value from the context (usually only on recompile, unfortunately).

This sniffed value is used for cardinality estimation and plan selection, but it is not used to simplify the query or otherwise depend on the specific parameter value. So, parameter sniffing can help pick a plan that is good for a specific case. Because most data sets have nonuniform column distributions, the value sniffed can affect the run time of the query plan. If a value representing the common distribution is picked, this might work very well in the average case and less optimally in the outlier case (a value with substantially more instances than the average case). If the outlier is used to sniff the value, then the plan picked might perform noticeably worse than it would have if the average case value had been sniffed. This can be a problem due to the plan caching policy in SQL Server—a parameterized query is kept in the cache even though the values change from execution to execution. When a recompile happens, only the information from that specific context is used to recompile.

The OPTIMIZE FOR hint allows the query author to specify the actual values to use during compilation. This can be used to tell the Query Optimizer, “This is a common value that I expect to see at run time,” and this can provide more plan predictability on parameterized queries. This hint works for both the initial compilation and for recompiles. While specifying a common value is usually the best approach, test out this hint to make sure that it gives the desired behavior.

In Listing 8-17, the OPTIMIZE FOR hint is used to force the query plan to account for an average value in the optimization of the query. (Note: I am only demonstrating that the plans change, not that these two plans perform differently. This technique can be used on arbitrarily complex queries to hint plans.) Note that when the value of 23 is used to compile the query (as shown in Figure 8-79), a different index is picked than when it is not (shown in Figure 8-78) because 23 is a very common value and it is not as selective as the predicate on *col2*. Parameter values can cause index changes, join order changes, and other more complex changes to your query plan—testing forced parameters is highly recommended.

Listing 8-17: Parameter Sniffing Example

```
CREATE TABLE param1(col1 INT, col2 INT);
go
SET NOCOUNT ON;
BEGIN TRANSACTION;
DECLARE @a INT=0;
WHILE @a < 10000
BEGIN
INSERT INTO param1(col1, col2) VALUES (@a, @a);
SET @a+=1;
END;
COMMIT TRANSACTION;
go
CREATE INDEX i1 ON param1(col1);
go
CREATE INDEX i2 ON param1(col2);
go
DECLARE @b INT;
DECLARE @c INT;
SELECT * FROM param1 WHERE col1=@b AND col2=@c;
```

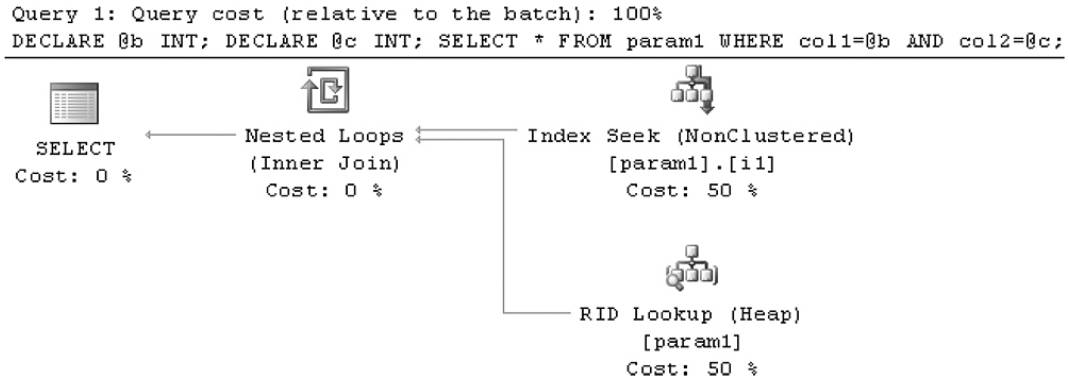


Figure 8-78: Non-sniffed parameters use index i1

```
SELECT * FROM param1 WHERE col1=23 AND col2=5;
```

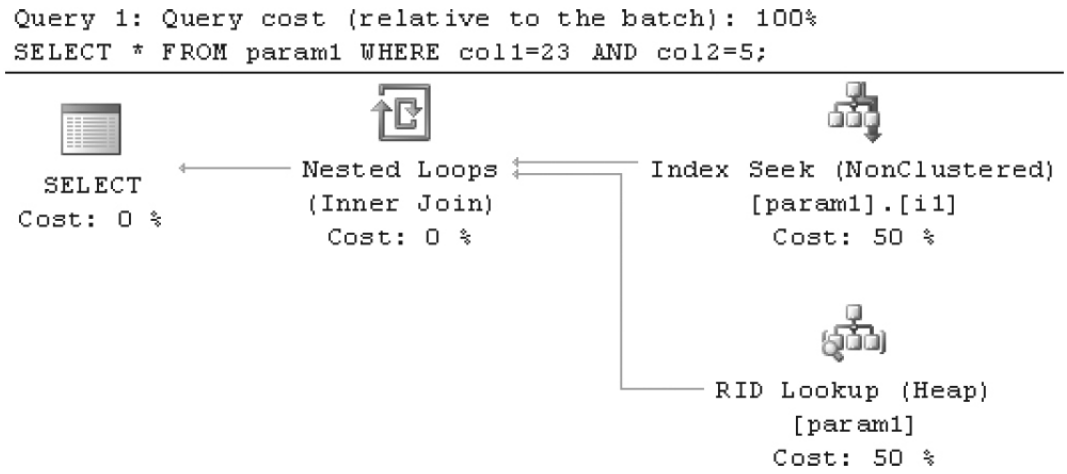


Figure 8-79: Sniffed parameters use i2

```
DECLARE @b INT;
DECLARE @c INT;
SELECT * FROM param1 WHERE col1=@b AND col2=@c
OPTION (OPTIMIZE FOR (@b=22));
```

Using the OPTIMIZE FOR hint instructs the Query Optimizer to use a known common value when generating the plan so that it works for a wide range of parameter values.

PARAMETERIZATION {SIMPLE | FORCED}

SIMPLE parameterization is the model that has existed in SQL Server for many releases. This corresponds to the concept of the trivial plan explained in this chapter. FORCED parameterization always replaces most literals in the query with parameters. As the plan quality can suffer, using FORCED should be done with care and an understanding of the global behavior of your application. Usually, FORCED mode should be used only in an OLTP system with many almost equivalent queries that (almost) always yield the same query plan. Essentially, you are betting that the plans will not change between possible parameter values. If all the queries are very small, the risk of this bet is smaller. The reasoning for this hint is that some OLTP systems with ad-hoc queries spent a large fraction of their time compiling the same (or similar) queries repeatedly. When possible, this is a good case to consider adding parameters into your application's queries.

NOEXPAND

By default, the query processor expands view definitions when parsing and binding the query tree. While the Query Optimizer usually matches the indexed views during optimization (as well as portions of any query even when the indexed view was not specified), there are some cases where the internal queries are rewritten such that it is not possible to match indexed views anymore. The NOEXPAND hint forces the query processor to force the use of the indexed view in the final query plan. In many cases, this can speed up the execution of the query plan because the indexed view often pre-computes an expensive portion of a query. However, this is not always true—the Query Optimizer may be able to find a better plan using the information from the fully expanded query tree.

USE PLAN

The USE PLAN *N'xml plan'* hint directs the Query Optimizer to try to generate a plan that looks like the plan in the supplied XML string. The Query Optimizer has been instrumented to use the shape of this plan as a series of hints to guide the optimization process to get the desired plan shape. Note that this does not guarantee that the _exact_ same plan is selected, but it will usually be identical or very close.

The common use of this hint is a DBA or database developer who wishes to fix a plan regression in the Query Optimizer. If a baseline of good/expected query plans is saved when the application is developed or first deployed, these can be used later to force a query plan to change back to what was expected if the Query Optimizer later determines to change to a different plan that is not performing well. This could be necessary to force a join order to avoid locking deadlocks or merely

to get the right physical plan shape and algorithms to be chosen. In some scenarios, the Query Optimizer does not have enough information to make a good decision about a portion of the query plan (for example, the join order) and it can lead to a suboptimal plan choice. DBAs should use this option with care—forcing the original query plan may actually degrade performance further because the plan was likely created for different data volumes and distributions. Try out a plan hint on a test database before deploying it, when possible.

Although this feature was added in SQL Server 2005, the feature has been improved in SQL Server 2008 with the inclusion of scripting support through Management Studio and the ability to hint more types of queries. For example, *INSERT/DELETE/UPDATE/MERGE* queries are now supported in USE PLAN hints, which can be very useful in forcing specific update plans that avoid deadlocks in stress scenarios.

While SQL Server 2008 supports additional query types, some are not supported with this feature. These include:

- Dynamic, Keyset, and Fast Forward cursors
- Queries containing remote tables
- Full-text Queries
- DDL commands, including *CREATE INDEX* and *ALTER PARTITION FUNCTION,* which manipulate data

In the context of rules, properties, and the Memo, the USE PLAN hint is used by the Query Optimizer to control both the initial shape of the query tree (for example, the initial join order after the tree is normalized early in Optimization) as well as the rules that are enabled to run for each group in the Memo. In the case of join orders, the Query Optimizer enables only join order transformations that led to the configuration specified in the plan hint. Physical implementation rules are also hinted, meaning that a hash aggregate in the XML plan hint requires that the implementation rule for hash aggregation be enabled and that the stream aggregation rule be disabled.

The following example demonstrates how to retrieve a plan hint from SQL Server and then apply it as a hint to a subsequent compilation to guarantee the query plan:

```
CREATE TABLE customers(id INT, name NVARCHAR(100));
CREATE TABLE orders(orderid INT, customerid INT, amount MONEY);
go
SET SHOWPLAN_XML ON;
go
SELECT * FROM customers c INNER JOIN orders o ON c.id = o.customerid;
```

The *SELECT* statement returns a single row and single column of text of XML that contains the XML plan for the query. It is too large to print in the book, but it starts with

```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="1.0"
...
```

Once we have copied the XML, it is necessary to escape single quotes before we can use it in the USE PLAN hint. Usually, I copy the XML into an editor and then search for single quotes and replace them with double quotes. Then we can copy the XML into the query using the OPTION (USE PLAN '<xml …/>') hint. (The hint was again shortened for space.)

```
SET SHOWPLAN_XML OFF;
SELECT  * FROM customers  c INNER JOIN orders o ON c.id = o.customerid
OPTION (USE PLAN '<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/
showplan" Version="1.0" . ..');
```

This technique makes it possible to force one query plan in scenarios when you can manipulate the query submitted to the server. The names used in the XML plan format are logical (table names instead of *object_id*), so it should be possible to take a USE PLAN hint from one table and use it on another with the same physical schema (columns, indexes, and so on) with only minor modifications. A very straightforward way to copy a plan from one table to another table with the same structure, or from one database or SQL Server instance to another is to create a *plan guide* incorporating the USE PLAN hint. Plan Guides are discussed in Chapter 9.

Summary

The Query Optimizer is a complex component with many internal features. While it is not always possible to know exactly why the Query Optimizer chose a specific plan, knowing a little about the Query Optimizer's design can help a DBA or database developer examine any query plan and diagnose any problems. Knowing how the Query Optimizer works can

also help reinforce good database design methodologies that can improve the quality of your application and reduce problems in deployment.

This chapter explains the mechanisms used in query processing and optimization, including trees, rules, properties, and the Memo framework. These ideas are used through different stages of optimization to try to find a reasonable plan quickly. The examples throughout the chapter demonstrate many of the operators and how they are used to implement the SQL queries submitted by the user. Finally, the use of the statistics profile output can help identify poorly optimized queries and to use statistics and hints to get the Query Optimizer to select a better plan.