Inside Microsoft SQL Server 2005: T-SQL Querying

by Itzik Ben-Gan, Lubor Kollar and Dejan Sarka

Microsoft Press. (c) 2006. Copying Prohibited.

Chapter 2: Physical Query Processing

Lubor Kollar

Overview

While the previous chapter described *what* outcome a query execution result should produce, this one will explain *how* Microsoft SQL Server 2005 attains that outcome.

The SQL language is spoken by most database experts, and all relational database products include some dialect of the SQL standard. Nevertheless, each product has its own particular query-processing mechanism. Understanding the way a database engine processes queries helps software architects, designers, and programmers make good choices when designing database schemas and writing queries.

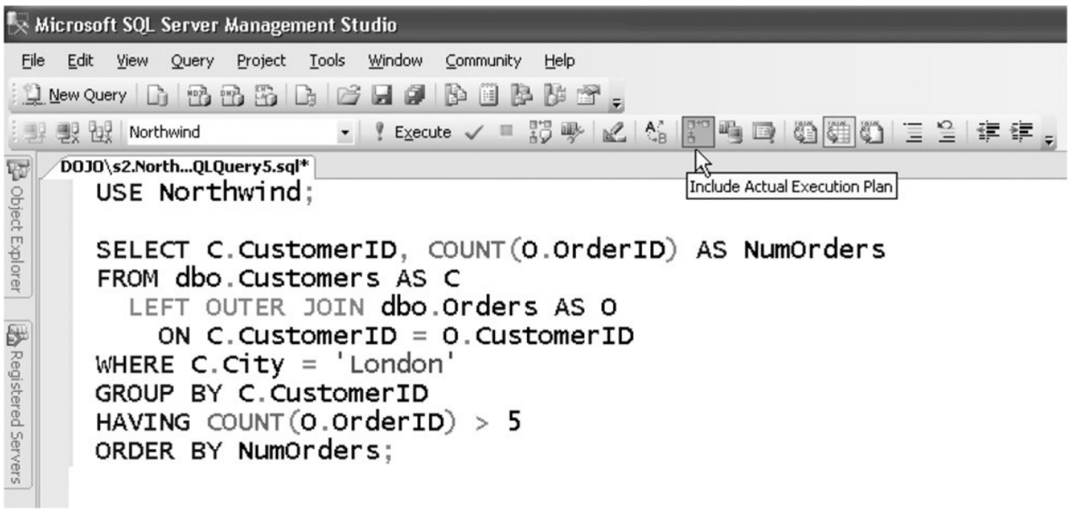
When a query reaches the database engine, the SQL Server performs two major steps to produce the desired query result. The first step is query compilation, which generates a *query plan*, and the second step is the execution of the query plan.

Query compilation in SQL Server 2005 consists of three steps: parsing, algebrization, and query optimization. After those steps are completed, the compiler stores the optimized query plan in the procedure cache. There, the execution engine copies the plan into its executable form and subsequently executes the steps in the query plan to produce the query result. If the same query or stored procedure is executed again and the plan is located in the procedure cache, the compilation step is skipped and the query or stored procedure proceeds directly to execution reusing the stored plan.

In this chapter, we will look at how the query optimizer produces the query plan and how you can get your hands on both the estimated and actual plans used when processing the query. This is a case where starting with an example of the final product, together with a description of how the product performs its desired function, helps us understand the process of building the product itself. Therefore, I will start with an example of executing a query that is similar to the one we worked with in Chapter 1. Then, once the basics are understood, I will look more closely inside the query compilation process and describe the various forms of query plans.

Flow of Data During Query Processing

If you want to make the upcoming example a hands-on experience, start SQL Server Management Studio (SSMS). Run the query shown in Listing 2-1 against the Northwind database, after clicking the Include Actual Execution Plan icon, as shown in Figure 2-1.



```
USE Northwind;

SELECT C.CustomerID, COUNT(O.OrderID) AS NumOrders
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.CustomerID = O.CustomerID
WHERE C.City = 'London'
GROUP BY C.CustomerID
HAVING COUNT(O.OrderID) > 5
ORDER BY NumOrders;
```

Figure 2-1: Include Actual Execution Plan option

Listing 2-1: Query against Northwind to demonstrate the result of optimization

```
USE Northwind;

SELECT C.CustomerID, COUNT(O.OrderID) AS NumOrders
```

```
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.CustomerID = O.CustomerID
WHERE C.City = 'London'
GROUP BY C.CustomerID
HAVING COUNT(O.OrderID) > 5
ORDER BY NumOrders;
```

Note The Northwind database is not shipped with SQL Server 2005. You can download the SQL Server 2000 version (which works on SQL Server 2005 as well) from http://www.microsoft.com/technet/prodtechnol/sql/2000/downloads/default.mspx. The installation script will create a directory named SQL Server 2000 Sample Databases on your C: drive, and there you will find instnwnd.sql. Run this script in SSMS to create the Northwind database. Alternatively, you can use CREATE DATABASE Northwind FOR ATTACH … to attach the NORTHWND.LDF and NORTHWND.MDF files to your instance of SQL Server 2005.

Note Remember that you can download the source code for the book from http://www.insidetsql.com.

You will obtain a graphical query plan similar to the one in Figure 2-2 in the Execution Plan pane of the SSMS window.

Note The execution plan in Figure 2-2 contains some elements that were added for demonstration purposes, and you will not see them in the plan that you will get. For each arrow, I have added the estimated numbers of rows in parentheses and a reference number used in the following text.

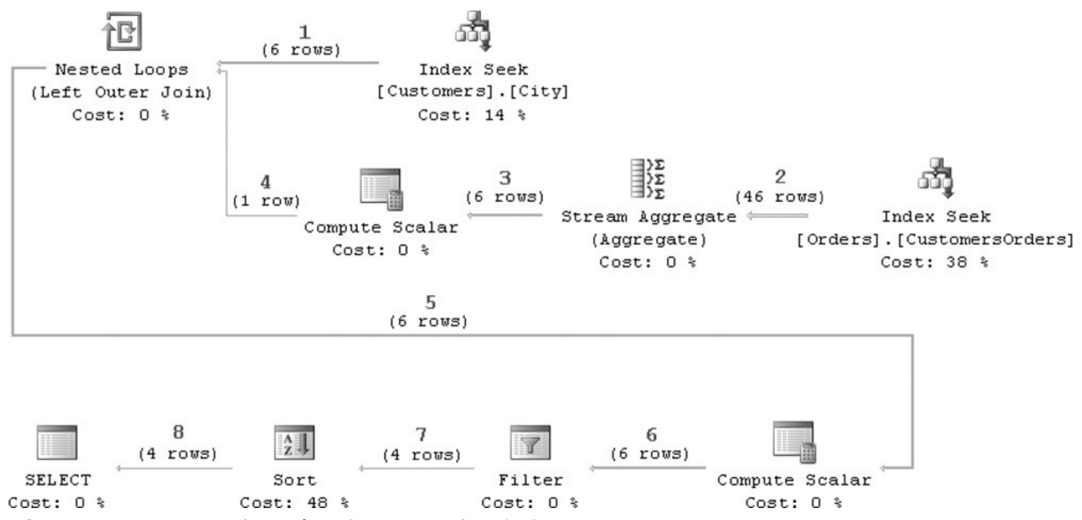


Figure 2-2: Execution plan for query in Listing 2-1

This query returns the ID (*CustomerID*) and number of orders placed (*NumOrders*) for all customers from London that placed more than five orders. The result is shown in Table 2-1.

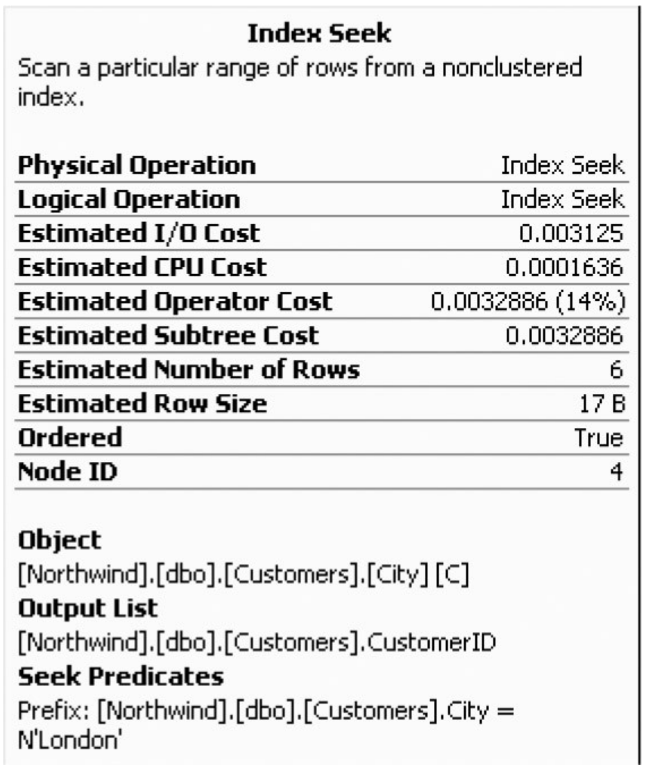
Table 2-1: Output of Query in Listing 2-1

CustomerID	NumOrders
EASTC	8
SEVES	9
BSBEV	10
AROUT	13

How does SQL Server execute the plan shown in Figure 2-2 to produce the desired result?

The execution of the individual branches is interleaved. I will demonstrate the process in an upcoming example, where SQL Server alternates its activity between the two branches of the Nested Loops step. To start, keep in mind that all gray arrows in Figure 2-2 represent data streams—rows produced by the operator are consumed by the next operator in the direction of the arrow. The thickness of the arrows corresponds to the relative number of rows the query optimizer is estimating will flow through the connection.

The engine starts execution by performing the Index Seek at the top of Figure 2-2 on the Customers table–and it will select the first row with the customer residing in London. You can see the seek predicate *Prefix: [Northwind].[dbo].[Customers].City = N'London'* in a small pop-up window if you hover the cursor over the Index Seek on the Customer table icon, as shown in Figure 2-3. The selected row is passed to the Nested Loops operator on the arrow 1, and as soon as it reaches the Nested Loops the so-called inner side of the Nested Loops operator is activated. In our case, in Figure 2-2 the inner side of the Nested Loops operator consists of the Compute Scalar, Stream Aggregate, and Index Seek operators connected to the Nested Loops by arrows 4, 3, and 2, respectively.



Index Seek
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001636
Estimated Operator Cost	0.0032886 (14%)
Estimated Subtree Cost	0.0032886
Estimated Number of Rows	6
Estimated Row Size	17 B
Ordered	True
Node ID	4

Object
[Northwind].[dbo].[Customers].[City] [C]
Output List
[Northwind].[dbo].[Customers].CustomerID
Seek Predicates
Prefix: [Northwind].[dbo].[Customers].City = N'London'

Figure 2-3: Pop-up information window for the Index Seek operator

If we investigate the Index Seek operator on the inner side of the Nested Loops in Figure 2-2, we find out that its seek predicate is *Prefix: [Northwind].[dbo].[Orders].CustomerID = [Northwind] .[dbo].[Customers].[CustomerID] as [C].[CustomerID]*. We see that the *C.CustomerID* value is used to seek into the Orders table to retrieve all orders for the *CustomerID*. This is an example where the inner side of the Nested Loops references the value obtained in the other, so-called outer side of the Nested Loops.

After all orders for the first London customer are fetched, they are passed via the arrow marked 2 to the Stream Aggregate operator, where they are counted and the resulting count named *Expr1004* is stored in the row by the Compute Scalar operator between arrows 3 and 4. Then the row composed from the *CustomerID* and the order count is passed through arrows 5 and 6 to the Filter operator with the predicate *[Expr1004] > (5)*. *Expr1004* represents the expression *COUNT (O.OrderID)*, and the *(5)* is the constant we have used in the query to limit the result to only customers with more than five orders.

Again, you can see the predicate in a pop-up window once you position the cursor on the Filter icon. If the predicate holds (meaning the customer has more than five orders), the row is passed to the Sort operator via arrow 7. Observe that SQL Server cannot output any rows from the Sort until it collects all the rows to be sorted. This is because the last row that arrives at the Sort potentially could be the one that should be "first" in the given order (the customer with the lowest number of orders exceeding five in our case). Therefore, the rows are "waiting" in the Sort, and the above outlined process is repeated for the next London customer found in the Index Seek on the Customers table. Once all the rows to be returned reach the Sort operator, it will return them in the correct order (arrow 8).

Compilation

A batch is a group of one or more Transact-SQL statements compiled as a single unit. A stored procedure is an example of a batch. Another example is a set of statements in the Query window in the SQL Pane in SSMS. The GO command divides sets of statements into separate batches. Observe that GO is not a T-SQL statement. SQLCMD, OSQL, and SSMS use the keyword GO to signal the end of a batch.

SQL Server compiles the statements of a batch into a single executable unit called an *execution plan*. During compilation, the compiler expands the statements by including the relevant constraints, triggers, and cascading actions that have to be carried out during the statement execution. If the compiled batch contains invocations of other stored procedures or functions and their plans are not in the cache, the stored procedures and functions are recursively compiled as well. The main steps in batch compilation are shown in Figure 2-4.

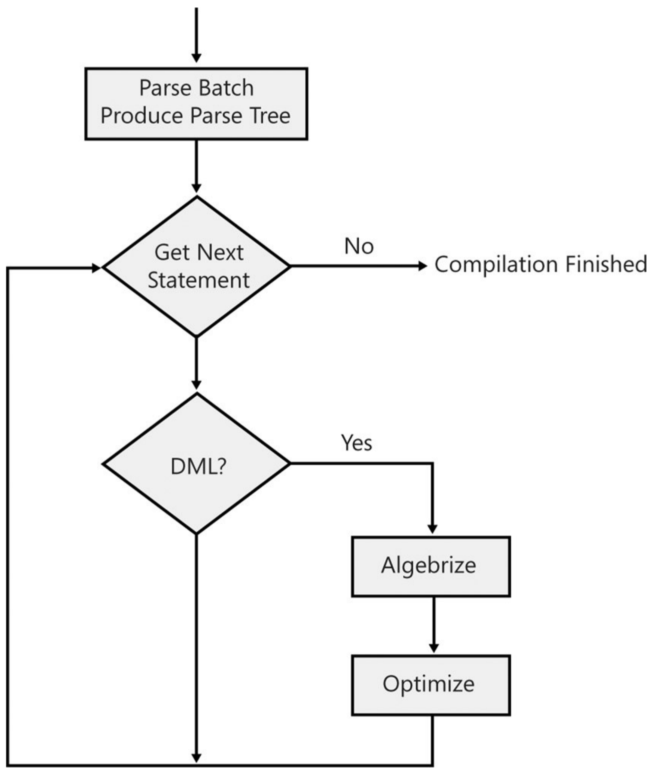


Figure 2-4: Compilation

It is important to be aware that compilation and execution are distinct phases of query processing and that the gap between when SQL Server compiles a query and when the query is executed can be as short as a few microseconds or as long as several days. An ad-hoc query usually doesn't have its plans in the cache when it is processed; therefore, it is compiled and its plan is immediately executed. On the other hand, the compiled plan for a frequently executed stored procedure might reside in the procedure cache for a very long time because SQL Server removes the infrequently used plans from the procedure cache first if the storage space is required for other purposes, including storing new query plans.

The optimizer is taking into account how many CPUs are available for SQL Server and the amount of memory that is available for query execution. However, the number of CPUs available to execute the query and the amount of available memory can change dramatically from moment to moment. You really have to think of compilation and execution as two separate activities, even when you are submitting an ad-hoc SQL statement through SSMS and executing it immediately.

When SQL Server is ready to process a batch, an execution plan for the batch might already be available in SQL Server's cache. If not, the compiler compiles the batch and produces a query plan. The compilation process encompasses a few things. First, SQL Server goes through the phases of *parsing* and *binding*. Parsing is the process of checking the syntax and transforming your SQL batch into a parse tree. Parsing is a generic operation used by compilers for almost all programming languages. The only specific thing in SQL Server's parser is its own grammar for defining valid T-SQL syntax.

Parsing includes checking, for example, whether a nondelimited table or column name starts with a digit. The parser flags an error if one is found. However, parsing does not check whether a column used in a WHERE clause really exists in any of the tables listed in the FROM clause; that issue is dealt with during binding.

The binding process determines the characteristics of the objects that you reference inside your SQL statements, and it checks whether the semantics you're asking for make sense. For example, while a query including FROM A JOIN B may be parsed successfully, binding will fail if A is a table and B is a stored procedure.

Optimization is the last step in the compilation. The optimizer has to translate the nonprocedural request of a set-based SQL statement into a procedure that can execute efficiently and return the desired results. Similar to binding, optimization is performed one statement at a time for all statements in the batch. After the compiler generates the plan for the batch and stores the plan in the procedure cache, a special copy of the plan's *execution context* is executed. SQL Server caches the execution contexts much like it does with the query plans, and if the same batch starts the second execution before the first one is finished, SQL Server will create the second execution context from the same plan. You can learn more about the SQL Server procedure cache from *Inside Microsoft SQL Server 2005: Query Tuning and Optimization* (Microsoft Press, 2006) by Kalen Delaney or from the white paper "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005" at http://www.microsoft.com/technet/prodtechnol/sql/2005/ recomp.mspx#EJAA.

SQL Server does not optimize every statement in the batch. It optimizes only certain classes of statements: those that access database tables and for which there might be multiple execution choices. SQL Server optimizes all DML (data manipulation language) statements—these are SELECT, INSERT, DELETE, and UPDATE statements. In addition to the DML, some other T-SQL statements are optimized; CREATE INDEX is one of them. Only the optimized statements will produce query plans. The following example shows that the optimizer creates a plan for CREATE INDEX:

```
CREATE TABLE dbo.T(a INT, b INT, c INT, d INT);
INSERT INTO dbo.T VALUES(1, 1, 1, 1);
SET STATISTICS PROFILE ON; -- forces producing showplan from execution
CREATE INDEX i ON dbo.T(a, b) INCLUDE(c, d);
SET STATISTICS PROFILE OFF; -- reverse showplan setting
DROP TABLE dbo.T; -- remove the table
```

It will produce this optimized query plan:

```
insert [dbo].[T] select *,%%bmk%% from [dbo].[T]
  |--Index Insert(OBJECT:([db].[dbo].[T].[i]))
    |--Sort(ORDER BY:([db].[dbo].[T].[a] ASC, [db].[dbo].[T].[b] ASC, [Bmk1000] ASC))
      |--Table Scan(OBJECT:([db].[dbo].[T]))
```

Similar to the CREATE INDEX statement, CREATE STATISTICS, UPDATE STATISTICS, and some forms of ALTER INDEX are also optimized. Several statements executed internally to perform database checking in DBCC CHECKDB are optimized as well. However, be aware that out of these non-DML optimized statements only CREATE INDEX produces a showplan with a statistics profile and none of them produces a query plan directly in SSMS. (Showplans will be explained later in the "Working with the Query Plan" section.)

Algebrizer

The *algebrizer*[*] is a new component in SQL Server 2005, and binding is its most important function. (Note that because the binding is the most significant function of the algebrizer often the whole process performed by the algebrizer is called *binding*). The algebrizer replaces the *normalizer* in SQL Server 2000. The algebrizer is a good example of the long-term focus of the SQL Server development team. With each release of SQL Server, several parts of the product are completely re-architected and rewritten to maintain a healthy code base. In the case of the algebrizer, the redesign spanned two releases of SQL Server. The development team's goals were not only to rewrite but also to completely redesign the logic to serve current and future expansions of SQL Server functionality.

The output of parsing—a parse tree—is the algebrizer's input. After performing several walks through the parse tree, the algebrizer produces its output—called a *query processor tree*—that is ready for query optimization.

In addition to binding, which is mostly concerned with name resolution by accessing the catalog information, the algebrizer flattens some binary operators and performs type derivation. In addition to performing name resolution, the algebrizer performs special binding for aggregates and groupings.

Operator Flattening

The algebrizer performs flattening of the binary operators UNION, AND, and OR. The parser's notion of these operators is binary only, as demonstrated on the left side of the illustration in the Figure 2-5 for the expression *(A=1) OR (A=2) OR (A=3) OR (A=4) OR (A=5) OR (A=6) OR (A=7)*. On the other hand, all compilation passes following the parser prefer to assemble multiples of binary operators into single *n*-ary operator whenever possible, as shown on the right side of the same figure. This is especially important for very long IN lists that the parser converts into chains of Ors. Among other things, flattening will eliminate most of the stack-overflow problems in subsequent passes that are caused by very deep trees. The code inside SQL Server that performs the flattening is itself carefully written to use iteration rather than recursion whenever possible so that the algebrizer itself is not prone to the same problem.

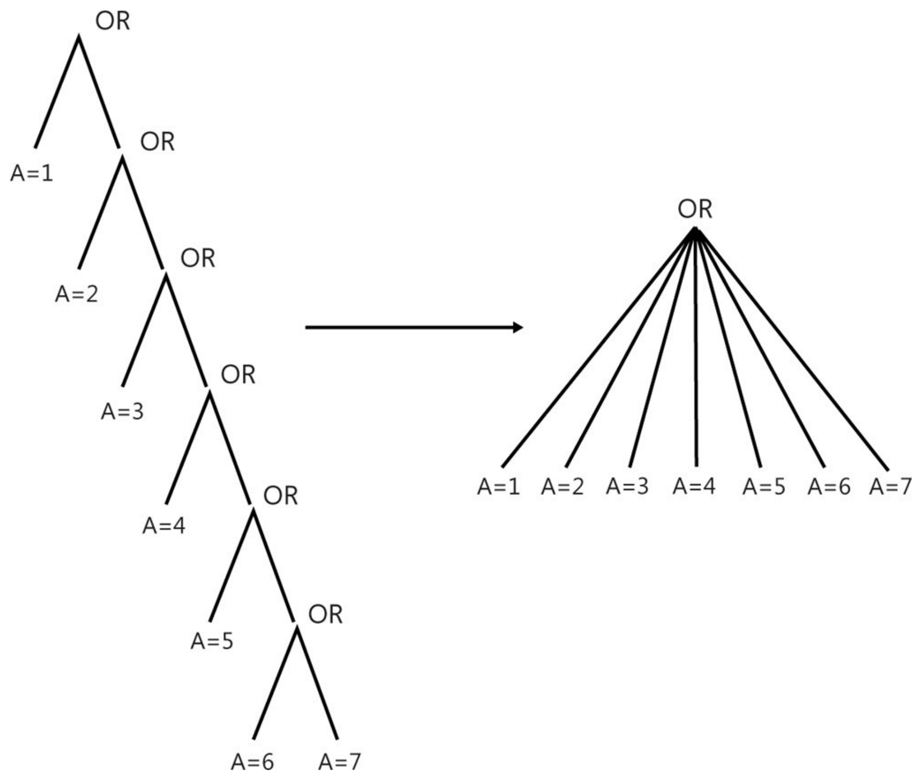


Figure 2-5: OR flattening

Name Resolution

Every table and column name in the parse tree is associated with a reference to the corresponding table or column definition object. Names representing the same object get the same reference. This is important information for the next step—query optimization. The algebrizer checks that every object name in the query actually refers to a valid table or column that exists in the system catalogs and is visible in the particular query scope. The algebrizer subsequently associates the object name with information from the catalogs.

Name resolution for views is the process of replacing a view reference by its view tree (the parse tree of the query that defines the view). Views are resolved recursively if they refer to additional views.

Type Derivation

Because T-SQL is statically typed, the algebrizer determines the type of each node of the parse tree. The algebrizer performs this in a bottom-up fashion, starting from the leaf nodes–columns (whose type information is in the catalogs) and constants. Then, for a non-leaf node, the type information is derived from the types of the children and the attributes of the node. A good example of type derivation is the process of figuring out the final data type of a UNION query, where different data types can appear in corresponding column positions. (See the "Guidelines for Using Union" chapter in the SQL Server 2005 Books Online for more details.)

Aggregate Binding

Consider the following example against table T1 with columns *c1* and *c2*, and table T2 with column *x* of the same data type as *T1.c2*:

```
SELECT c1 FROM dbo.T1
GROUP BY c1
HAVING EXISTS
  (SELECT * FROM dbo.T2
   WHERE T2.x > MAX(T1.c2));
```

SQL Server computes the MAX aggregate in the outer query *SELECT c1 FROM dbo.T1 GROUP BY c1*, although it is syntactically located in the inner one. The algebrizer makes this decision based on the aggregate's argument. The bottom line is that every aggregate needs to be bound to its hosting query–the place where it is correctly evaluated. This is known as *aggregate binding*.

Grouping Binding

This is perhaps the least obvious of all activities performed by the algebrizer. Let's consider an example against table T1 with columns *c1*, *c2*, and *c3*:

```
SELECT c1 + c2, MAX(c3) FROM dbo.T1 GROUP BY c1 + c2;
```

The use of the expression *c1 + c2* in the SELECT list is legitimate, although had we placed just *c1* in this list, the query would have been semantically incorrect. The reason for this is that *grouped queries* (for example, those with an explicit GROUP BY or HAVING clause) have different semantics than nongrouped ones. In particular, all nonaggregated columns or expressions in the SELECT list of a query with GROUP BY must have a direct match in the GROUP BY list, and the process of verifying this fact is known as *grouping binding*.

Unfortunately, explicit clauses such as GROUP BY or HAVING aren't the only factors that might force a SELECT list to become grouped. According to SQL's rules, just the presence of an aggregate function that binds to a particular list makes that SELECT list grouped, even if it has no GROUP BY or HAVING clauses. Here is a simple example:

```
SELECT c1, MAX(c2) FROM dbo.T1;
```

This is a grouped SELECT, because there is a MAX aggregate. Because it is a grouped SELECT, the use of the nonaggregated column *c1* is illegal and the query is incorrect.

An important role of the algebrizer is to identify any semantic errors in the statement. The following example shows that this is a nontrivial task for some queries with aggregates:

```
SELECT c1, (SELECT T2.y FROM dbo.T2 WHERE T2.x = MAX(T1.c2)) FROM dbo.T1;
```

This query is incorrect for the same reason as the previous one, but it is clear that we have to complete aggregate binding for the entire query just to realize this. The *MAX(T1.c2)* in the inner query must be evaluated in the outer query much as it was in the query *SELECT c1, MAX(c2) FROM dbo.T1;* just shown, and therefore, the use of the nonaggregated column *c1* in the SELECT list is illegal and the query is incorrect.

Optimization

One of the most important and complex components involved in processing your queries is the query optimizer. The optimizer's job is to produce an efficient execution plan for each query in a batch or a stored procedure. The plan lists the steps SQL Server has to carry out to execute your query, and it includes such information as which index or indexes to use when accessing data from each table in the query. The plan also includes the strategy for processing each join operation, each aggregation, each sort, and each partitioned table access. The plan shows an intent to perform operations on parallel threads–that is, where the row streams are partitioned, repartitioned, and then merged into a single stream.

SQL Server's query optimizer is a cost-based optimizer, which means that it tries to come up with the cheapest execution plan for each SQL statement. The cost of the plan reflects the estimated time to complete the query. For each query, the optimizer must analyze the possible plans and choose the one with the lowest estimated cost. Some complex statements have millions of possible execution plans. In these cases, the query optimizer does not analyze all possible combinations. Instead, it tries to find an execution plan that has a cost reasonably close to the theoretical minimum. Later in this section, I'll explain some ways the optimizer can reduce the amount of time it spends on optimization.

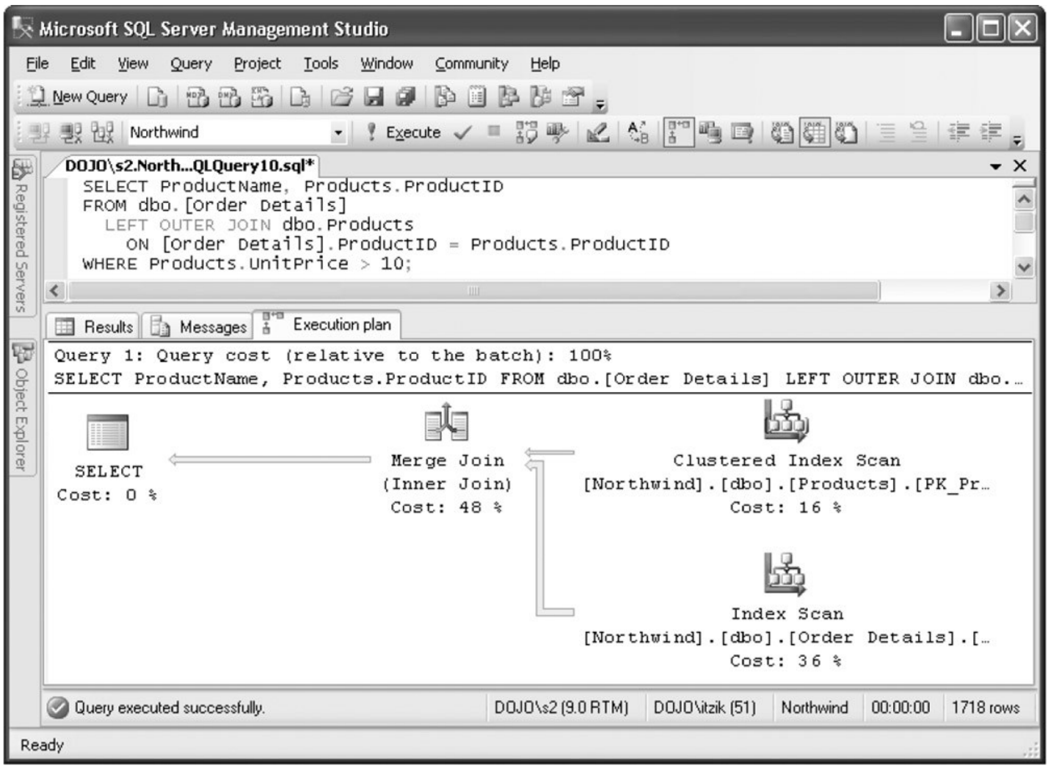
The lowest estimated cost is not necessarily the lowest resource cost; the query optimizer chooses the plan that most quickly returns results to the user with a reasonable cost in resources. For example, processing a query in parallel (using multiple CPUs simultaneously for the same query) typically uses more resources than processing it serially using a single CPU, but the query completes much faster in parallel. The optimizer will propose a parallel execution plan to return results, and SQL Server will use such a parallel plan for execution if the load on the server is not adversely affected.

Optimization itself involves several steps. The *trivial plan* optimization is the first step. The idea behind trivial plan optimization is that cost-based optimization is expensive to initialize and run. The optimizer can try many possible variations in looking for the cheapest plan. If SQL Server knows by investigating the query and the relevant metadata that there is only one viable plan for a query, it can avoid a lot of the work required to initialize and perform costbased optimization. A common example is a query that consists of an INSERT with a VALUES clause into a table that does not participate in any indexed views. There is only one possible plan. Another example is a SELECT from single table with no indexes and no

GROUP BY. In these two cases, SQL Server should just generate the plan and not try to find something better. The trivial plan the optimizer finds is the obvious plan, and usually it is very inexpensive. Later in the chapter, I will show how to determine whether the optimizer produced a trivial plan for a particular query.

If the optimizer doesn't find a trivial plan, SQL Server will perform some simplifications, which are usually syntactic transformations of the query itself, to look for commutative properties and operations that can be rearranged. SQL Server can perform operations that don't require considering the cost or analyzing what indexes are available but that result in a more efficient query. An example of simplification is to evaluate simple single table where filters before the joins. As described in Chapter 1, the filters are *logically* evaluated after the joins, but evaluating the filters before the joins produces correct result as well and is always more efficient because it removes unqualified rows before the join operation.

Another example of simplification is transforming outer joins into inner joins in some cases, as shown in Figure 2-6. In general, an outer join will add rows to the result set of an inner join. These additional rows have the NULL value in all columns of the inner set if there is no inner row satisfying the join predicate. Therefore, if there is a predicate on the inner set that disqualifies these rows, the result of the outer join is the same as the inner join and it will never be cheaper to generate the outer join result. Therefore, SQL Server changes the outer join to an inner join during simplification. In the following OUTER JOIN query, the predicate *Products.UnitPrice > 10* disqualifies all additional rows that would be produced by the OUTER JOIN, and therefore, the OUTER JOIN is simplified into an INNER join:



```
SELECT ProductName, Products.ProductID
FROM dbo.[Order Details]
  LEFT OUTER JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice > 10;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT ProductName, Products.ProductID FROM dbo.[Order Details] LEFT OUTER JOIN dbo.…
```

Figure 2-6: Outer join simplification

```
USE Northwind;
SELECT
  [OrderDetails].OrderID,
  Products.ProductName,
  [Order Details].Quantity,
  [Order Details].UnitPrice
FROM dbo.[Order Details]
  LEFT OUTER JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice > 10;
```

SQL Server then loads up the statistical information on the indexes and tables, and the optimizer begins the cost-based optimization process.

The cost-based optimizer is using a set of transformation rules that try various permutations of data access strategies, join orders, aggregation placement, subquery transformations, and other rules that guarantee the correct result is still produced. Usually, correct results are the same results; however, it is important to recognize that for some queries there is more than one correct result. For example, any set of 10 orders would be a correct result for the query

```
SELECT TOP (10) <select_list> FROM Orders;
```

A change to data that affects the optimizer's cost-based analysis could change the chosen plan, which in turn could change the result unexpectedly.

If the optimizer compared the cost of every valid plan and chose the least costly one, the optimization process could take a very long time—the number of valid plans can be huge. Therefore, the optimization is broken up into three *search phases*. A set of transformation rules is associated with each phase. After each phase, SQL Server evaluates the cost of the cheapest query plan to that point. If the plan is cheap enough, SQL Server ends the optimization and chooses that plan. If the plan is not cheap enough, the optimizer runs the next phase, which contains an additional set of usually more complex rules.

Many queries, even though they are complicated, have very cheap plans. If SQL Server applied many transformation rules and tried various join orders, the optimization process could take substantially longer than the query execution itself. Therefore, the first phase of the costbased optimization, Phase 0, contains a limited set of rules and is applied to queries with at least four tables. Because join reordering alone generates many potential plan candidates, the optimizer uses a limited number of join orders in Phase 0, and it considers only Hash joins and Nested Loops. If this phase finds a plan with an estimated cost below 0.2, the optimization ends. The queries with final query plans produced by Phase 0 are typically found in transaction processing applications; therefore, this phase is also called the *Transaction Processing phase*.

The next step, Phase 1 or Quick Plan optimization, uses more transformation rules and tries different join orders. When the phase is completed, if the best plan costs less than 1.0, the optimization ends. Up to this point, the optimizer considers only nonparallel query plans. If more than one CPU is available to SQL Server and the least expensive plan produced by Phase 1 is more than the *cost threshold for parallelism* (which you can determine by using sp_configure to find the current value—the default is 5), Phase 1 is repeated with the goal of finding the best parallel plan. The costs of the serial and parallel plans obtained in Phase 1 are compared, and Phase 2, the Full Optimization phase, is executed for the cheaper of the two. Phase 2 contains additional rules—for example, it uses Outer Join reordering and automatic Indexed View substitution for multitable views. Figure 2-7 shows the phases of query optimization in SQL Server 2005.

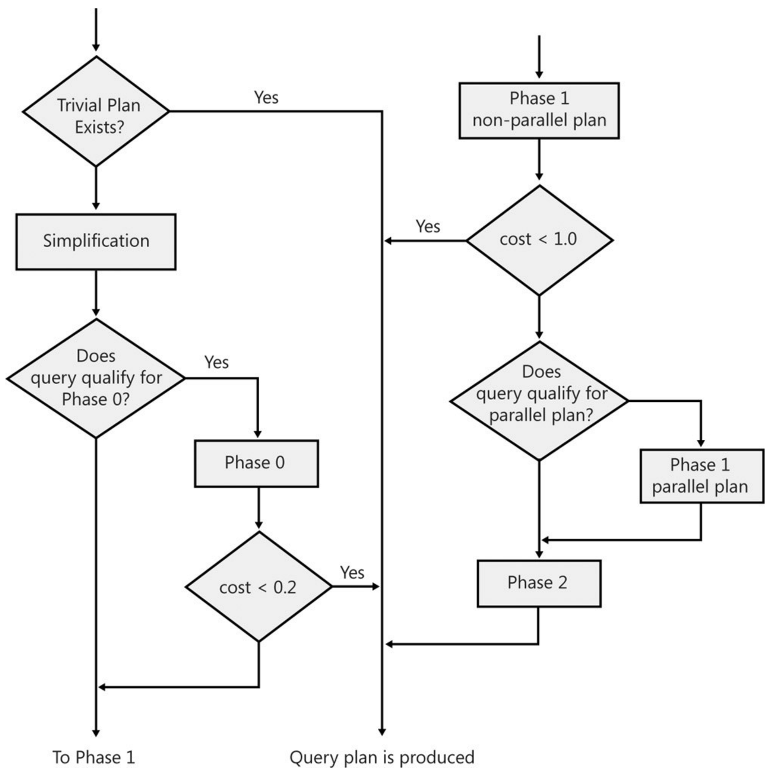


Figure 2-7: Phases of query optimization

SQL Server 2005 provides plenty of insight into its own operation. This is true also for query optimization. In the previous releases of SQL Server, the only product of query optimization was the query plan; the history of the optimization was forgotten, and only the final result was preserved. In SQL Server 2005, there exists a new peephole into the optimizer's activity—it is the dynamic management view (DMV) sys.dm_exec_query_optimizer_info. This view provides cumulative information about all the optimizations performed since the SQL Server was started.

Using this DMV, you can find out what optimizer events are happening while the optimizer is processing your batches. The sys.dm_exec_query_optimizer_info DMV returns three columns: *counter*, *occurrence*, and *value*. The column named *counter* provides the name of the optimizer event. The *occurrence* column shows the cumulative number of occurrences of the optimizer event, and some events are using the *value* column to provide additional event-specific values. For example, each time the optimizer chooses a trivial plan, the *occurrence* column value for the trivial plan counter will be increased by one. Similarly, you can find out how many times each optimization phase—Phase 0, 1, or 2—was executed by investigating the corresponding "search 0", "search 1", or "search 2" events. The *value* column is used, for example, for the "tables" event—it captures the average number of tables referenced in the optimized statements. Please refer to the sys.dm_exec_query_optimizer_info topic in the "SQL Server Language Reference" section of Books Online for a detailed description of all counters returned by the sys.dm_exec_query_optimizer_info DMV.

When using sys.dm_exec_query_optimizer_info, you should be careful about the procedure cache. If the cache already contains the plan for your query or batch, the optimization phase is skipped and there will be no optimizer events generated. You can use DBCC FREEPROCCACHE to clear up the procedure cache to ensure the compilation will take place afterwards. However, you should be careful using the DBCC FREEPROCCACHE on production servers because it will delete the contents of the procedure cache, and all statements and stored procedures will have to be compiled anew.

Because the counters of optimizer events are cumulative, you want to find their values before and after the optimization of your batch or workload if you are interested in the generated events and statistics. However, execution of ‘select * from sys.dm_exec_query_optimizer_info’ itself might generate optimizer events. Therefore, I have carefully constructed the code example in Listing 2-2 to avoid self-spoiling of the optimizer information provided by the sys.dm_exec_query_optimizer_info in SQL Server 2005. Your statement or batch has to be inserted in the marked place inside the example in Listing 2-2. Out of the 38 types of events in this DMV, the example will display only those that either have changed the *occurrence* or *value* column as a result of your statement or batch. After you insert your code, you should execute the whole batch once from SSMS to obtain the optimizer events triggered by your code.

Listing 2-2: Script to obtain information about your batch from sys.dm_exec_query_optimizer_info

```
SET NOCOUNT ON;
USE Northwind; -- use your database name here
DBCC FREE PROCCACHE; -- empty the procedure cache
GO
-- we will use tempdb..OptStats table to capture
-- the information from several executions
-- of sys.dm_exec_query_optimizer_info
IF (OBJECT_ID('tempdb..OptStats') IS NOT NULL)
  DROP TABLE tempdb..OptStats;
GO
-- the purpose of this statement is
-- to create the temporary table tempdb..OptStats
SELECT 0 AS Run, *
INTO tempdb..OptStats
FROM sys.dm_exec_query_optimizer_info;
GO
-- this will populate the procedure cache
-- with this statement's plan so that it will not
-- generate any optimizer events when executed
-- next time
-- the following GO is intentional to ensure
-- the query plan reuse will happen for the following
-- INSERT for its next invocation in this script
GO
INSERT INTO tempdb..OptStats
  SELECT 1 AS Run, *
  FROM sys.dm_exec_query_optimizer_info;
GO
-- same reason as above; observe the "2" replaced "1"
-- therefore, we will have a different plan
GO
```

```
INSERT INTO tempdb..OptStats
  SELECT 2 AS Run, *
  FROM sys.dm_exec_query_optimizer_info;
GO
-- empty the temporary table
TRUNCATE TABLE tempdb..OptStats
GO
-- store the "beforerun" information
-- in the temporary table with the output
-- of sys.dm_exec_query_optimizer_info
-- with value "1" in the column Run
GO
INSERT INTO tempdb..OptStats
  SELECT 1 AS Run, *
  FROM sys.dm_exec_query_optimizer_info;
GO
-- your statement or batch is executed here
/*** the following is an example
SELECT C.CustomerID, COUNT(O.OrderID) AS NumOrders
FROM dbo.Customers AS C
  LEFT OUTER JOIN dbo.Orders AS O
    ON C.CustomerID = O.CustomerID

WHERE C.City = 'London'
GROUP BY C.CustomerID
HAVING COUNT(O.OrderID) > 5
ORDER BY NumOrders;
***/
GO
-- store the "after run" information
-- in the temporary table with the output
-- of sys.dm_exec_query_optimizer_info
-- with value "2" in the column Run
GO
INSERT INTO tempdb..OptStats
  SELECT 2 AS Run, *
  FROM sys.dm_exec_query_optimizer_info;
GO
-- extract all "events" that changed either
-- the Occurrence or Value column value between
-- the Runs 1 and 2 from the temporary table.
-- Display the values of Occurrence and Value
-- for all such events before (Run1Occurrence and
-- Run1Value) and after (Run2Occurrence and
-- Run2Value) executing your batch or query.
-- This is the result set generated by the script.
WITH X (Run, Counter, Occurrence, Value)
AS
(
  SELECT *
  FROM tempdb..OptStats WHERE Run=1
),
Y (Run, Counter, Occurrence, Value)
AS
(
  SELECT *
  FROM tempdb..OptStats
  WHERE Run=2
)
SELECT X.Counter, Y.Occurrence-X.Occurrence AS Occurrence,
  CASE (Y.Occurrence-X.Occurrence)
    WHEN 0 THEN (Y.Value*Y.Occurrence-X.Value*X.Occurrence)
        ELSE (Y.Value*Y.Occurrence-X.Value*X.Occurrence)/(Y.Occurrence-X.Occurrence)
  END AS Value
FROM X JOIN Y
  ON (X.Counter=Y.Counter
      AND (X.Occurrence<>Y.Occurrence OR X.Value<>Y.Value));
GO
-- drop the temporary table
```

```
DROP TABLE tempdb..OptStats;
GO
```

If we use the preceding script to investigate the compiler events and the corresponding counters and values for the statement from Listing 2-1, after we run the script with the statement embedded in the marked place (–your statement or batch is executed here), we will see the result of the statement itself followed by Table 2-2. From the counters, we see that SQL Server optimized the batch consisting of a single statement 0.008752 seconds, the cost of the final plan is 0.023881, the DOP is 0 (which means serial plan), a single optimization has been executed, only Phase 1 (same as "search 1") of optimization has been exercised using 647 search tasks in 0.00721 seconds (the "search" is almost always the most expensive part of the query compilation), and the query has 2 tables. If the batch consists of multiple statements, the *Value* column in Table 2-2 would contain average values of the counters and statistics.

Table 2-2: Counters of the Optimizer Event for the Statement from Listing 2-1

Counter	Occurrence	Value
Elapsed time	1	0.008752
Final cost	1	0.023881
Maximum DOP	1	0
Optimizations	1	1
search 1	1	1
search 1 tasks	1	647
search 1 time	1	0.00721
Tables	1	2
Tasks	1	647

Working with the Query Plan

Showplan is the term used by SQL Server users to name the textual, graphical, or XML form of a query plan produced by the query optimizer. We use it also as a verb to name the process of obtaining the query plan. A showplan shows information about how SQL Server will (or did) process the query. For each table in the query plan, a showplan tells whether indexes are used or whether a table scan is necessary. It also indicates the order of execution of the different operations in the plan. Reading showplan output is as much an art as it is a science, but it really just takes a lot of practice to get comfortable interpreting it. I hope that the description and examples in this section will be enough to give you a good start.

Note Throughout the book, you will find discussions where query plans are analyzed; therefore, both this chapter and the next spend a fair amount of space describing how to work with query plans and how to analyze them. This chapter will teach you how to obtain various forms of the query plans, while Chapter 3 will teach you how to examine those from a query-tuning perspective. You will find some overlap of content in both chapters, but the discussions are vital to providing a thorough background for the rest of the book.

SQL Server 2005 can produce showplans in any of three different formats: graphical, text, and XML. When considering the content, SQL Server can produce plans with operators only, plans with additional cost estimates, and plans with additional run-time information. Table 2-3 summarizes the commands and interfaces used to obtain the query plans with different content in the various formats:

Table 2-3: Commands Generating Various Formats of a Showplan

	Format		
Content	Text	XML	Graphical
Operators	SET SHOWPLAN_TEXT ON	N/A	N/A
Operators and estimated costs	SET SHOWPLAN_ALL ON	SET SHOWPLAN_XML ON	Display Estimated Execution Plan in Management Studio

Run-time info	SET STATISTICS PROFILE ON	SET STATISTICS XML ON	Include Actual Execution Plan in Management Studio

Let's start with the simplest forms of the showplan.

SET SHOWPLAN_TEXT and SHOWPLAN_ALL

Here's an example of SHOWPLAN_TEXT for a two-table join from the Northwind database:

```
SET NOCOUNT ON;
USE Northwind;
GO
SET SHOWPLAN_TEXT ON;
GO
SELECT ProductName, Products.ProductID
FROM dbo.[Order Details]
  JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice > 100;
GO
SET SHOWPLAN_TEXT OFF;
GO
```

The following is the result of executing the preceding code in SQL Server Management Studio:

```
StmtText
---------------------------------------------------------------------------
SELECT ProductName, Products.ProductID
FROM dbo.[Order Details]
  JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice> 100;

StmtText
---------------------------------------------------------------------------

  |--Nested Loops(Inner Join, OUTER REFERENCES: ([Northwind].[dbo].[Products].[ProductID]))
       |--Clustered Index Scan(OBJECT: ([Northwind].[dbo].[Products].[PK_Products]),
WHERE:([Northwind].[dbo].[Products].[UnitPrice]>($100.0000)))
       |--Index Seek(OBJECT:([Northwind].[dbo].[Order Details].[ProductID]),
SEEK:([Northwind].[dbo].[Order Details].[ProductID]= [Northwind].[dbo].[Products]
.[ProductID]) ORDERED FORWARD)
```

The output tells us the query plan consists of three operators: Nested Loops, Clustered Index Scan, and Index Seek. The Nested Loops is performing an inner join on the two tables. The outer table (the table accessed first when performing the inner join, which is always the one on the upper branch entering the join operator) in the join is the Products table, and SQL Server is using a Clustered Index Scan to access the physical data. Because a clustered index contains all the table data, scanning the clustered index is equivalent to scanning the whole table. The inner table is the Order Details table, which has a nonclustered index on the *ProductID* column, and SQL Server is using an Index Seek to access the index rows. The Object reference after the Index Seek operator shows us the full name of the index used: [Northwind].[dbo].[Order Details].[ProductID]. In this case, it is a bit confusing because the name of the index is the same as the name of the join column. For this reason, I recommend not giving indexes the same names as the columns they will access. The Seek predicate follows the Object reference. The outer table's column value is used to perform the seek into the ProductID index.

When the plan is executed, the general flow of the rows is from the top down and from right to left. The more indented operator produces rows consumed by the less indented operator, and it produces rows for the next operator above, and so forth. In the case of a join, there are two input operators at the same level to the right of the join operator denoting the two joined row sets. The higher of the two (the Clustered Index Scan in our example) is referred to as the outer table, and the lower (Index Seek in our example) is the inner table. The operation on the outer table is initiated first; the one on the inner table is repeatedly executed for each row of the outer table that arrives to the join operator. In addition to the binary join operators, there are also *n*-ary operators with *n* input branches–for example, concatenation in the plans for queries with UNION ALL. For the *n*-ary operators, the upper branch is executed first, and then the lower branch, and so forth.

Observe that in the preceding example I used SET SHOWPLAN_TEXT OFF following the query. This is because SET SHOWPLAN_TEXT ON is not only causing the query plan to show up, it is also turning off query execution for the connection. Query execution will stay turned off until SQL Server executes SET SHOWPLAN_TEXT OFF on the same connection. You must be careful when performing a showplan of a batch that is creating or altering indexes or permanent tables and subsequently using them in queries in the same batch. Because SQL Server does not execute the batch if SHOWPLAN_TEXT is turned on, the new or altered objects in the batch are not recognized when subsequently referenced. Consequently, you will see a failure if a permanent table is created and used in the same batch, or you might incorrectly think the newly created index will not be used in the query plan. The only exceptions to this rule are temporary tables and table variables that are created to produce the showplan, but their creation is subsequently rolled back at the end of the showplan execution.

SHOWPLAN_ALL is very similar to SHOWPLAN_TEXT. The only difference is the additional information about the query plan produced by SHOWPLAN_ALL. It adds estimates of the number of rows produced by each operator in the query plan, the estimated size of the result rows, the estimated CPU time, and the total cost estimate that was used internally when comparing this plan to other possible plans. I won't show you the output from SHOWPLAN_ALL because it's too wide to fit nicely on a page of this book. But the returned information is still only a subset of the information compared to the XML format of a showplan, which I describe next.

XML Form of the Showplan

There are two different kinds of XML showplans. One, obtained through SET SHOWPLAN_XML ON, contains an estimated execution plan; the second one, the output of the SET STATISTICS XML ON, includes run-time information as well. Because the output of SET SHOWPLAN_XML is generated by a compilation of a batch, it will produce a single XML document for the whole batch. On the other hand, the output of SET STATISTICS XML is produced at runtime and you will see a separate XML document for each statement in the batch. In addition to the SET commands, there are two other ways to obtain the XML showplan—by saving the graphical showplan displayed in SSMS, and by using the SQL Server Profiler. I will describe both of them later in this chapter.

A single XML schema, showplanxml.xsd, covers both the estimated and run-time XML show-plans; however, the run-time output provides additional information. Therefore, its specific elements and attributes are optional in the xsd. Installing SQL Server 2005 places the schema in the Microsoft SQL Server\90\Tools\Binn\schemas\sqlserver\2004\07\showplan directory. It is also available at http://www.schemas.microsoft.com/sqlserver.

A showplan in XML format can be saved in a file and given the extension sqlplan (for example, batch1.sqlplan). Opening a file with a .sqlplan will automatically use SQL Server Management Studio (if installed) to display a graphical showplan. There is no need to connect to the server where the showplan was produced or which holds the referenced objects. This great new feature of SQL Server 2005 enables working with the stored and shared (for example, through e-mail) showplans in graphical form without permanently storing large static images.

XML is the richest format of the showplan. It contains some unique information not available in any other textual or graphical showplan. For example, only the XML showplan contains the size of the plan (the *CachedPlanSize* attribute) and parameter values for which the plan has been optimized (the *ParameterList* element), and only the run-time XML showplan contains the number of rows processed in different threads of a parallel plan (the *ActualRows* attribute of the *RunTimeCountersPerThread* element) or the true degree of parallelism when the query was executed (the *DegreeOfParallelism* attribute of the plan).

As I explained earlier, a single XML showplan can capture information about several statements in a batch. You should have this in mind when developing software that processes the XML showplan. You should definitely think about cases of multistatement batches and include them in your tests unless you will be processing only XML showplan documents that are produced by the SET STATISTICS XML or by SQL Server Profiler.

Probably the greatest benefit of the XML format is that it can be processed using any XML technology—for example, XPath, XQuery, or XSLT.

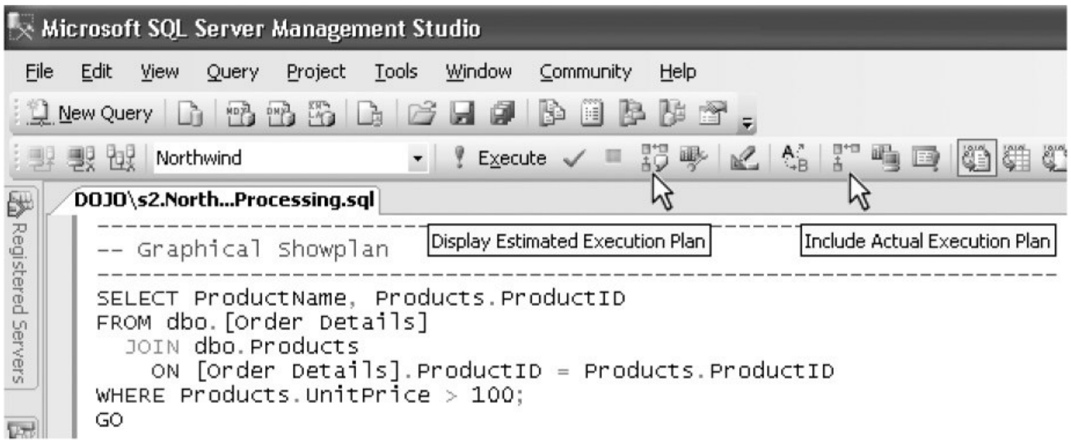
More Info A good example of extracting data from the XML showplan can be found at http://www.msdn.microsoft.com/library/default.asp?url=/ library/en-us/dnsql90/html/xmlshowplans.asp.

This white paper describes an application that extracts the estimated execution cost of a query from its XML showplan. Using this technique, a user can restrict the submitting of queries to only queries that cost less than a predetermined threshold. This will ensure that long-running queries will not overload the server.

I'm convinced the XML showplan will lead to the development of numerous tools to help the administrators, programmers, and operational personnel with their daily work. New queries that help to analyze the XML showplans are appearing on the Internet with increasing frequency.

Graphical Showplan

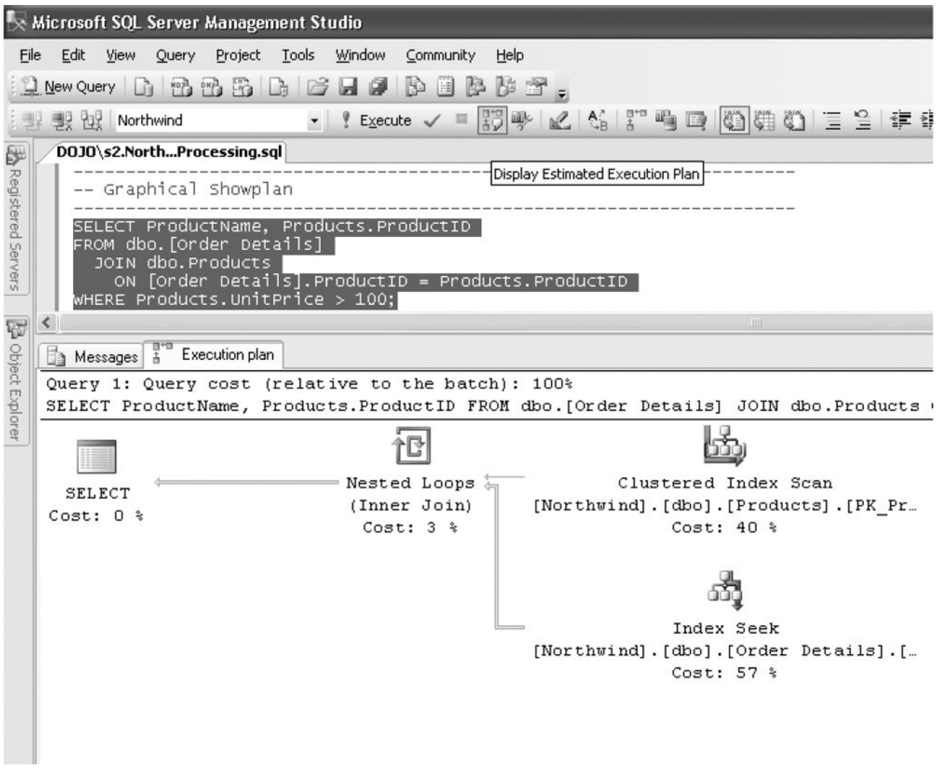
SSMS has two options to present the graphical form of a showplan: the Display Estimated Execution Plan and Include Actual Execution Plan commands, which are available in the Query menu and as toolbar buttons as depicted in Figure 2-8.


Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Community Help
New Query
Northwind
Execute
DOJO\s2.North...Processing.sql
Display Estimated Execution Plan
Include Actual Execution Plan
Registered Servers

```
-----------------------------------------------------------------------
-- Graphical Showplan
-----------------------------------------------------------------------
SELECT ProductName, Products.ProductID
FROM dbo.[Order Details]
  JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice > 100;
GO
```

Figure 2-8: Display Estimated Execution Plan and Include Actual Execution Plan options in SSMS

There is a significant difference between the Display Estimated Execution Plan and Include Actual Execution Plan options. If you select the former, a picture showing the graphical showplan of the query or batch in the query window is shown almost immediately (with the speed depending on the compilation time and whether the plan is already cached or not) under the Execution Plan tab in the result portion of SSMS, as shown in Figure 2-9.

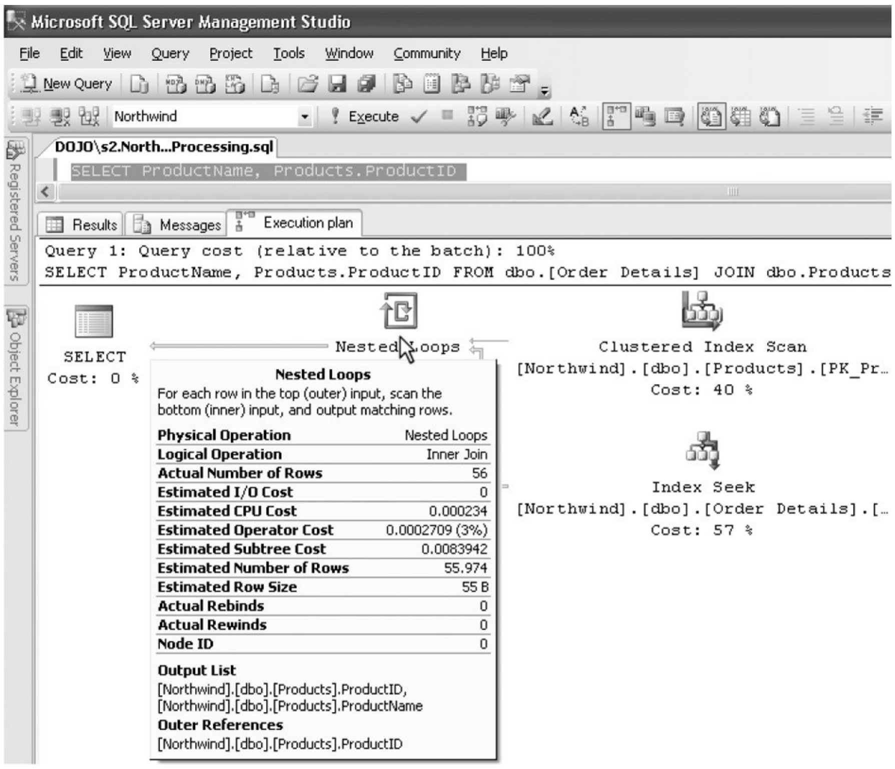

Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Community Help
New Query
Northwind
Execute
DOJO\s2.North...Processing.sql
Display Estimated Execution Plan
Registered Servers
Object Explorer
Messages
Execution plan
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 3 %
Clustered Index Scan
[Northwind].[dbo].[Products].[PK_Pr…
Cost: 40 %
Index Seek
[Northwind].[dbo].[Order Details].[…
Cost: 57 %

```
-----------------------------------------------------------------------
-- Graphical Showplan
-----------------------------------------------------------------------
SELECT ProductName, Products.ProductID
FROM dbo.[Order Details]
  JOIN dbo.Products
    ON [Order Details].ProductID = Products.ProductID
WHERE Products.UnitPrice > 100;
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT ProductName, Products.ProductID FROM dbo.[Order Details] JOIN dbo.Products
```

Figure 2-9: Display Estimated Execution Plan

Activating the Include Actual Execution Plan button has no immediate action associated with it. It only changes the state of SSMS to include the run-time showplan with any executed statement or batch. I activated the button, and then executed the same query as above by clicking the Execute button with the red exclamation mark. The additional Results tab appeared among the result windows alongside the Messages and Execution Plan tabs in the lower half of the SSMS window. If you select the tabs one by one, you will see that the Results window contains the query result and the Execution Plan window has a plan that is a similar plan to the one just shown in Figure 2-9. The differences in the plan become apparent only after you investigate the contents of the information inside the individual operators. In Figure 2-10, I hovered the mouse pointer over the Nested Loops operator, which brought up additional information that is not present in the Estimated Plan—fields and values for "Actual Number Of Rows," "Actual Rebinds," and "Actual Rewinds" (all of which will be explained later).


Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Community Help
New Query
Northwind
Execute
DOJO\s2.North...Processing.sql
Registered Servers
Object Explorer
Results
Messages
Execution plan
SELECT
Cost: 0 %
Nested Loops
Clustered Index Scan
[Northwind].[dbo].[Products].[PK_Pr...
Cost: 40 %
Index Seek
[Northwind].[dbo].[Order Details].[...
Cost: 57 %

```
SELECT ProductName, Products.ProductID
```

```
Query 1: Query cost (relative to the batch): 100%
SELECT ProductName, Products.ProductID FROM dbo.[Order Details] JOIN dbo.Products
```

Nested Loops

For each row in the top (outer) input, scan the bottom (inner) input, and output matching rows.

Physical Operation	Nested Loops
Logical Operation	Inner Join
Actual Number of Rows	56
Estimated I/O Cost	0
Estimated CPU Cost	0.000234
Estimated Operator Cost	0.0002709 (3%)
Estimated Subtree Cost	0.0083942
Estimated Number of Rows	55.974
Estimated Row Size	55 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	0

Output List
[Northwind].[dbo].[Products].ProductID,
[Northwind].[dbo].[Products].ProductName
Outer References
[Northwind].[dbo].[Products].ProductID

Figure 2-10: Include Actual Execution Plan

The rows flow from the right to the left, and when joining two tables, the outer table is above the inner table in the graphical showplan.

If you right-click in the Execution Plan window, you will see a pop-up window with the choices shown in Figure 2-11. I used the Zoom To Fit option to produce the pictures of the plans in Figures 2-9 and 2-10. The Properties option will display properties of the operator you select prior to the right-click, similar to the example shown for the Nested Loops operator in Figure 2-10. Probably the least obvious action is associated with the Save Execution Plan As option. If you make this choice, SSMS prompts you for a file location in which to store the XML showplan (not the graphical representation of the plan). The default extension for the file name is sqlplan.

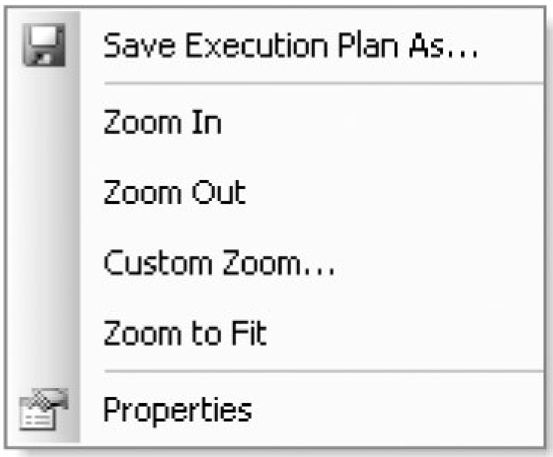


Figure 2-11: Execution plan options

Run-Time Information in Showplan

SQL Server collects the run-time information during the query execution. As I explained previously, XML showplan is the most complete form of a query plan. Therefore, we will start examining the additional run-time information returned in the XML showplan first. Then we will talk about the output of the SET STATISTICS PROFILE command, and finally, we'll discuss the SQL Server Profiler's run-time showplan information.

SET STATISTICS XML ON|OFF There are two kinds of run-time information in the XML showplan: per SQL statement and per thread. If a statement has a parameter, the plan contains the *ParameterRuntimeValue* attribute, which shows the value for each parameter when the statement is executed. This might differ from the value used to compile the statement under the *ParameterCompiledValue* attribute, but this attribute is in the plan only if the optimizer knows the value of the parameter at the optimization time and is true only for parameters passed to stored procedures.

Next, we have the *DegreeOfParallelism* attribute, which shows the actual degree of parallelism (or DOP, which is the number of concurrent threads working on the single query) of the execution. This, again, might be different from the compile-time value. The compile-time value is not captured in the query plan, but it is always equal to half the number of processors available to SQL Server unless the number of processors is 2–in which case, the compile-time DOP value will be 2 as well. The optimizer is considering half of the CPUs because the DOP at the execution time will be adjusted based on the workload at the time the execution starts; it might end up being any number between 1 and the number of processors. Regardless of the final choice for DOP, the same parallel plan is used. If a parallel plan ends up being executed with DOP = 1, SQL Server will remove the Exchange operators from the query plan when creating the execution context. The *MemoryGrant* attribute shows actual memory given to the query for execution in kilobytes. SQL Server uses this memory to build the hash tables for hash joins or to perform a sort in the memory.

The element *RunTimeCountersPerThread* contains five attributes, each with one value per thread: *ActualRebinds*, *ActualRewinds*, *ActualRows*, *ActualEndofScans*, and *ActualExecutions*. Chapter 3 describes how SSMS shows Actual Number Of Rows and Actual Rebinds And Actual Rewinds in the Operator information ToolTip box of a graphical showplan. The ToolTip box shows cumulative (added across all executions of all threads) values for each of the *ActualRows*, *ActualRebinds*, and *ActualRewinds* values from the XML showplan. The *Actual-Executions* value tells us how many times the operator has been initialized on each of the threads. If the operator is a scan operator, the *ActualEndofScans* count shows how many times the scan reached the end of the set. Consequently, subtracting *ActualEndofScans* from *Actual-Executions* tells us how many times the operator didn't scan the whole set–this might happen, for example, if TOP in the SELECT restricts the number of returned rows and the output set is collected before the scan reaches the end of the table. Similarly, in the case of a Merge Join, if one of the sets is exhausted we don't need to continue scanning the other set because there cannot be any more matches.

The XML showplan might also contain warnings. These are events generated either during compilation or during execution. Examples of compiler-generated warnings are missing statistics and a missing join predicate. Examples of run-time warnings are a hash bailout and an exchange spill. If you encounter a warning in your query plan, you should consult the "Errors and Warnings Event Category" in Books Online to find more information.

SET STATISTICS PROFILE SET STATISTICS PROFILE ON returns information similar to SET SHOWPLAN_ALL ON. There are two differences, however. SET STATISTICS PROFILE is active during statement execution, which is when it produces additional information alongside the actual result of the query. (The statement must finish execution before the additional output is produced.) Also, it adds two columns to the output: *Rows* and *Executes*. These values are derived from the run-time XML showplan output. The *Rows* value is the sum of the *Row-Count* attribute in the element *RunTimeCountersPerThread* across all threads. (There are multiple threads only if it is a parallel plan and if it was run in parallel.) The *Executes* value is the sum of the *ActualExecutions* attribute in the same element. Table 2-4 shows an example of a trimmed portion of the SET STATISTICS PROFILE output for the same query we were working with earlier.

Table 2-4: Output of STATISTICS PROFILE

Rows	Executes	StmtText
56	1	SELECT ProductName, Products.ProductId
56	1	|–Nested Loops(Inner Join, OUTER REFERENCES:([Northwind]
2	1	|–Clustered Index Scan(OBJECT:([Northwind].[dbo].[Products].[
56	2	|–Index Seek(OBJECT:([Northwind].[dbo].[Order Details].[Produ

The column *Rows* contains the number of rows actually returned by each operator. The number for *Executes* tells us how many times SQL Server initialized the operator to perform work on one or more rows. Because the outer side of the join (the Clustered Index Scan of the Products table) returned two rows, we had to execute the inner side of the join (the Index Seek) two times. Therefore, the Index Seek has the number 2 in the *Executes* column in the output.

When examining the plan for a particular query, a good way to find potential problems is to find the biggest discrepancies between the optimizer's estimates and the real number of executes and returned rows. Here we must be careful because the optimizer's estimate in the column *EstimateRows* is per estimated execution, while the *Rows* in the showplan output mentioned previously is the cumulative number of rows returned by the operator from all its executions. Therefore, to assess the optimizer's discrepancy we must multiply the *EstimateRows* by *EstimateExecutions* and compare the result with the actual number of all rows returned in the *Rows* column of the SET STATISTICS PROFILE output.

In some cases, it is possible to generate plans using SET STATISTICS PROFILE and SET STATISTICS XML for statements for which the SET SHOWPLAN does not produce any output. A CREATE INDEX on a nonempty table, sp_executesql, and a batch that creates and references the same object are such examples.

Capturing Showplan with SQL Trace

SQL Server Profiler is a GUI tool that defines and captures SQL Server 2005 trace events received from a server. SQL Server displays the events in the Profiler window and optionally saves them in a trace file or a table that can later be analyzed or used to replay a specific series of steps when trying to diagnose a problem. In Chapter 3, we explain why using the T-SQL code to define the trace is more efficient than using the Profiler GUI, and why T-SQL should be used when tracing nontrivial workloads. The following information about capturing the showplan is equally applicable to using both the GUI and the T-SQL sp_trace_ group of stored procedures.

Using a trace to capture the showplan information is very accurate because you will avoid rare but possible discrepancies between the showplan you examine in the SSMS and the actual plan used during the execution of your application. Even if you are not changing the metadata (adding or dropping indexes, modifying constraints, creating or updating statistics), you might still encounter cases where you have a different plan at execution time than at the original compile time. The most common cases occur when one stored procedure is called with different parameter values, when statistics are auto-updated, and when there is a change in available resources (CPU and memory) between compile time and run time. However, monitoring with a trace is resource intensive, and it might adversely affect the performance of your server. The more events you monitor, the bigger the impact. Therefore, you should choose the events you monitor carefully and alternatively consider extracting showplans from the procedure cache, as I will describe later. The downside of using the procedure cache is its volatility (if the query is not executed, the query plan might be removed from the cache) and the lack of run-time information about individual query executions.

There are nine event classes that capture various forms of showplan information in the Performance Event Category. Table 2-5 will help you to pick the most appropriate event for your needs.

Table 2-5: Showplan-Related Trace Event Classes

Trace Event Class	Compile or Run	Includes run-time info	Includes XML showplan	Generates trace against SQL Server 2000
Showplan All	Run	No	No	Yes
Showplan All for Query Compile	Compile	No	No	No
Showplan Statistics Profile	Run	Yes[1]	No	Yes
Showplan Text	Run	No	No	Yes
Showplan Text (Unencoded)	Run	No	No	Yes[2]
Showplan XML	Run	No	Yes	No
Showplan XML for Query Compile	Compile	No	Yes	No
Showplan XML Statistics Profile	Run	Yes[3]	Yes	No
Performance Statistics	Compile and Run [4]	Yes[5]	Yes	No

[1]The generated run-time information is identical to that produced by the SET STATISTICS PROFILE ON.

[2]If the SQL Server 2005 Profiler is connected to a SQL Server 2000 server, it automatically shows only the showplan events supported by

SQL Server 2000. Note that the "Showplan Text (Unencoded)" event is named "Execution plan" in SQL Server 2000.

[3]The *TextData* column in the profiler output contains the XML showplan, including all the same run-time information as in the output of SET STATISTICS XML ON.

[4]The Performance Statistics event class is a combination of several subevents. Subevents 1 and 2 produce the same showplan as the Showplan XML For Query Compile event—1 for stored procedures, and 2 for ad-hoc statements. Subevent 3 produces cumulative run-time statistics for the query when it is removed from the procedure cache.

[5]This is cumulative run-time information for all executions of this query produced for subevent 3. The captured information is the same as that produced by the sys.dm_exec_query_stats DMV, and it is stored in XML format in the *TextData* column in the trace output.

For simplicity, I will restrict further discussion to tracing against SQL Server 2005 only.

If your server is not very busy or is a development or test machine, you should use the Showplan XML Statistics Profile event. It generates all the query plan and run-time information you might need. You can also process the plans programmatically or display them in graphical form.

Even if your server is busy but the compile rate is kept low by good plan reuse, you can use the Showplan XML For Query Compile event because it produces trace records only when a stored procedure or statement is compiled or recompiled. This trace will not contain run-time information.

The Showplan Text event generates output identical to that of the Showplan Text (Unencoded) event except the showplan is stored as a character string in the *TextData* column for the latter plan and as encoded binary data in the *BinaryData* column for the former. A consequence of using the Binary Data column is that it's not human-readable—we need the Profiler or at least the Profiler's decoding DLL to read the showplan. The binary format also limits the number of levels in the XML document to 128. On the other hand, it requires less space to hold all the plan information.

You can reduce the size of the trace by selecting filter values for various columns. A good approach is to generate a small experimental trace, analyze which events and columns you don't need, and remove them from the trace. Using this process, you can also identify some missing information you might want to add to the trace. When setting up the trace filter (for example, by using sp_trace_setfilter), only the filters on the *ApplicationName*, *ClientProcessID*, *HostName*, *LoginName*, *LoginSid*, *NTDomainName*, *NTUserName*, and *SPID* columns suppress the generation of the event. The rest of the filters are applied only after the event has been generated and marshaled to the client. Therefore, you should not substitute extensive filtering for careful event selection; the filtering might potentially result in removing all records of a particular event class from the trace without helping to minimize the impact on the server. In fact, more work rather than less will be needed.

Compared to the SET STATISTICS PROFILE and SET STATISTICS XML options, the showplan trace events further enlarge the set of statements SQL Server captures the plans for. The additional classes of statements are auto- and non-auto-CREATE and UPDATE STATISTICS, as well as the INSERT INTO … EXEC statement.

Extracting the Showplan from the Procedure Cache

After the query optimizer produces the plan for the batch or stored procedure, the plan is placed in the procedure cache. You can examine the procedure cache using several dynamic management views (DMV) and functions (DMF), DBCC PROCCACHE, and the deprecated catalog view sys.syscacheobjects. I will show how you can access a showplan in XML format for the queries currently in the procedure cache.

The sys.dm_exec_query_plan DMF returns the showplan in XML format for any query whose query execution plan currently resides in the procedure cache. The sys.dm_exec_query_plan DMF requires a plan handle as its only argument. The plan handle is a VARBINARY (64) identifier of the query plan, and the sys.dm_exec_query_stats DMV returns it for each query currently in the procedure cache. The following query returns the XML showplans for all cached query plans. If a batch or stored procedure contains multiple SQL statements with a query plan, the view will contain a separate row for each one of them.

```
SELECT qplan.query_plan AS [Query Plan]
FROM sys.dm_exec_query_stats AS qstats
 CROSS APPLY sys.dm_exec_query_plan(qstats.plan_handle) AS qplan;
```

It is difficult to find the query plan for a particular query using the preceding query because the query text is contained only deep inside the XML showplan. The following extension to the previous query extracts the sequence number (the column named No) and the query text (the Statement Text column) from the showplan using the *Xquery* value method. Each batch has a single sql_handle; therefore, specifying *ORDER BY sql_handle, [No]* ensures the output rows for the batch containing multiple SQL statements are displayed one after another in the order they appear in the batch.

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/showplan' AS sql)
SELECT
  C.value('@StatementId','INT') AS [No],
  C.value('(./@StatementText)','NVARCHAR(MAX)') AS [Statement Text],
  qplan.query_plan AS [Query Plan]
FROM (SELECT DISTINCT plan_handle FROM sys.dm_exec_query_stats) AS qstats
  CROSS APPLY sys.dm_exec_query_plan(qstats.plan_handle) AS qplan
  CROSS APPLY query_plan.nodes('/sql:ShowPlanXML/sql:BatchSequence/sql:Batch/sql:Statements/
descendant::*[attribute::StatementText]')
    AS T(C)
ORDER BY plan_handle, [No];
```

Next, I will show a portion of the result returned by the preceding query. The output depends on the current contents of the procedure cache; therefore, your output will almost certainly be different than the one shown in Table 2-6.

Table 2-6: Information about Query Plans Extracted from Cache

No	Statement Text	Query Plan
1	SELECT CAST(serverproperty(N'S	<ShowPlanXML xmlns="http://sch
1	select value_in_use from sys.c	<ShowPlanXML xmlns="http://sch
1	with XMLNAMESPACES ('http://sc	<ShowPlanXML xmlns="http://sch
1	with XMLNAMESPACES ('http://sc	<ShowPlanXML xmlns="http://sch
1	IF (@@microsoftversion/0×010	<ShowPlanXML xmlns="http://sch
2	SELECT se.is_admin_endpoint A	<ShowPlanXML xmlns="http://sch
3	ELSE	<ShowPlanXML xmlns="http://sch
1	SELECT CAST(serverproperty(N'S	<ShowPlanXML xmlns="http://sch

Update Plans

The query optimizer must take care of several specific issues when optimizing INSERT, UPDATE, and DELETE—or, in other words, *data modifying*—statements. Here I will describe the techniques employed by the SQL Server to process these statements.

The IUD (shorthand I will use for "INSERT, UPDATE, and DELETE") plans have two stages. The first stage is *read only*, and it determines which rows need to be inserted/updated/deleted by generating a data stream describing the changes to be made. For INSERTs, the data stream contains column values; for DELETEs, it has the table keys; and for UPDATEs, it has both the table keys and the values of changed columns. The second stage applies changes in the data stream to the table; additionally, it takes actions necessary to preserve data integrity by performing constraint validation, it maintains nonclustered indexes and indexed views, and it fires triggers if they exist. Usually the UPDATE and DELETE query plans contain two references to the target table: the first reference is used to identify the affected rows and the second to perform the change. The INSERT plans contain only one reference to the target table unless the same target table also participates in generating the inserted rows.

In some simple cases, SQL Server merges the read and write stages of the IUD plans together. This is the case, for example, when inserting values directly into a table (a process known as a *scalar insert*) or updating/deleting rows identified by a value of a primary key on the target table.

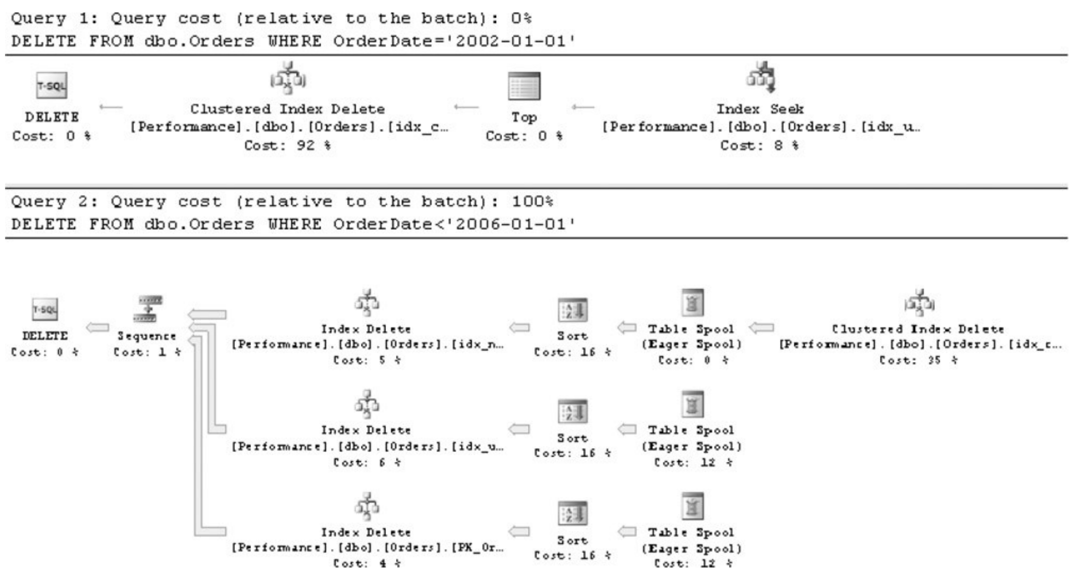
The Assert operator is automatically included in the query plans in the second phase if SQL Server needs to perform constraint validation. SQL Server validates the CHECK constraints for INSERTs and UPDATEs by evaluating a usually inexpensive scalar expression on each affected row and column. Foreign key constraints are enforced on INSERTs and UPDATEs to the table containing the foreign key constraint, and they're enforced on UPDATEs and DELETEs to the table containing the referenced key. The related table that is not the target of the IUD operation is scanned to verify the constraint; therefore, data access is involved. Declaring a primary key automatically creates a unique index on the key columns, but this is not the case for a foreign key. UPDATEs and DELETEs of referenced keys must access the foreign key table for each updated or deleted primary key value, either to validate nonexistence of the removed key or to propagate the change if it is a cascading referential integrity constraint. Therefore, you should ensure there is an index on the foreign key if you plan to perform UPDATEs affecting the key values or DELETEs from the primary table.

In addition to performing the IUD operation on the clustered index or heap, processing of the INSERT and DELETE queries also maintains all nonclustered indexes, and the UPDATE queries maintain indexes containing the modified columns. Because nonclustered indexes include the clustered index and partitioning keys to allow efficient access to the table row, updating columns that participate in the clustered index or in the partitioning key is expensive because it requires maintenance of all indexes. Updating the partitioning key might also cause rows to move between partitions. Therefore, when you have a choice, choose clustering and partitioning keys that you don't plan to update.

> **Note** SQL Server 2005 restricts the partitioning keys to a single column; therefore, "partitioning key" and "partitioning column" are synonyms.

In general, the performance of IUD statements is closely tied to the number of maintained indexes that include the target columns, because those must all be modified. Performing single-row INSERT and DELETE operations to an index requires a single index-tree traversal. SQL Server implements update to an index or partitioning key as a DELETE followed by an INSERT–therefore, it is roughly twice as expensive as a nonkey UPDATE.

The query optimizer considers and costs two different strategies for IUD statements: per-row and per-index maintenance. These two strategies are demonstrated in the plans for queries 1 and 2, respectively, in Figure 2-12. With per-row maintenance, SQL Server maintains the indexes and the base table together for each row affected by the query. The updates to all nonclustered indexes are performed in conjunction with each single row update on the base table (which might be a heap or a clustered index). The plan for Query 1 *DELETE FROM dbo.Orders WHERE OrderDate='2002-01-01'* (the top one in Figure 2-12) is an example of a per-row maintenance. Query 1is deleting only 24 rows in the Orders table in the Performance database that is generated by Listing 3-1 from Chapter 3. The plan for Query 1 does not show any deletes performed on the secondary indexes because they are carried out together with the deletes of the clustered index rows one row at a time.



```
Query 1: Query cost (relative to the batch): 0%
DELETE FROM dbo.Orders WHERE OrderDate='2002-01-01'
```

DELETE Cost: 0 %
Clustered Index Delete [Performance].[dbo].[Orders].[idx_c... Cost: 92 %
Top Cost: 0 %
Index Seek [Performance].[dbo].[Orders].[idx_u... Cost: 8 %

```
Query 2: Query cost (relative to the batch): 100%
DELETE FROM dbo.Orders WHERE OrderDate<'2006-01-01'
```

DELETE Cost: 0 %
Sequence Cost: 1 %
Index Delete [Performance].[dbo].[Orders].[idx_n... Cost: 5 %
Sort Cost: 16 %
Table Spool (Eager Spool) Cost: 0 %
Clustered Index Delete [Performance].[dbo].[Orders].[idx_c... Cost: 35 %
Index Delete [Performance].[dbo].[Orders].[idx_u... Cost: 6 %
Sort Cost: 16 %
Table Spool (Eager Spool) Cost: 12 %
Index Delete [Performance].[dbo].[Orders].[PK_Or... Cost: 4 %
Sort Cost: 16 %
Table Spool (Eager Spool) Cost: 12 %

Figure 2-12: Per-row and per-index update plans

The plan for Query 2 *DELETE FROM dbo.Orders WHERE OrderDate<'2006-01-01'* in the bottom portion of the Figure 2-12 will delete 751,216 rows against the same Orders table, and its plan is very different because it is performing per-index maintenance. At first, the plan deletes the qualifying rows from the clustered index (indicated by the Clustered Index Delete icon on the right) while at the same time building a temporary spool table containing the key values for the three nonclustered indexes that must be maintained. SQL Server reads the spooled data three times, once for each of these indexes. Between reading the spooled data and deleting the rows from the nonclustered index, SQL Server sorts the data in the order of the maintained index, thus ensuring optimal access to the index pages.

The Sequence operator enforces the execution order of its branches. SQL Server updates the indexes one after another from the top of the plan to the bottom.

The per-row update strategy is efficient in terms of CPU because there is a short code path required to update the table and all indexes together. The code for per-index maintenance is somewhat more complicated, but there might be significant savings in I/O. By updating the nonclustered indexes individually after sorting the keys, we will never visit an index page more than once even if many rows are updated on the same page. Therefore, a per-index update plan is usually chosen when many rows are updated and the optimizer estimates the same page of the maintained index would be read more than once to accomplish the maintenance using the per-row strategy.

In addition to the spools holding the keys for index maintenance, you might also encounter in the IUD plans the special spool operator that provides "Halloween protection" known also as the *Halloween spool*. (I will explain the origins of this name later in this chapter.) The query optimizer injects a spool operator into some IUD plans to ensure correctness of the produced result. I will use the following small example to demonstrate the problem. My Tiny_employees table has two columns—*name* and *salary*—and one nonclustered index on the *name* column to start with. Run the code in Listing 2-3 to create and populate the Tiny_employees table.

Listing 2-3: Script for Halloween database

```
SET NOCOUNT ON;
USE master;
GO
IF DB_ID('Halloween') IS NULL
  CREATE DATABASE Halloween;
GO
USE Halloween;
GO

-- Creating and Populating the Tiny_employees Table
IF OBJECT_ID('dbo.Tiny_employees') IS NOT NULL
  DROP TABLE dbo.Tiny_employees;
GO
CREATE TABLE dbo.Tiny_employees (name CHAR(8), salary INT);
INSERT INTO dbo.Tiny_employees VALUES ('emp_A',30000);
INSERT INTO dbo.Tiny_employees VALUES ('emp_B',20000);
INSERT INTO dbo.Tiny_employees VALUES ('emp_C',19000);
INSERT INTO dbo.Tiny_employees VALUES ('emp_D',8000);
INSERT INTO dbo.Tiny_employees VALUES ('emp_E',7500);
GO
CREATE INDEX ind_name ON dbo.Tiny_employees(name);
GO
```

Now consider implementing the following request: Increase salary by 10 percent for all employees with salary less than 25,000. The query is simple:

```
UPDATE dbo.Tiny_employees
  SET salary = salary * 1.1
WHERE salary < 25000;
```

Its query plan, shown in Figure 2-13, is using per-row maintenance. (You don't see an update node for the index ind_name anywhere in the plan.)

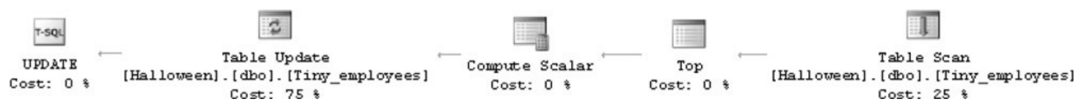


Figure 2-13: Execution plan for the UPDATE statement

Now let's create a clustered index on the Tiny_employees table on the *salary* column:

```
CREATE CLUSTERED INDEX ind_salary ON dbo.Tiny_employees(salary);
```

Again, let's investigate the query plan, shown in Figure 2-14, for the same query:

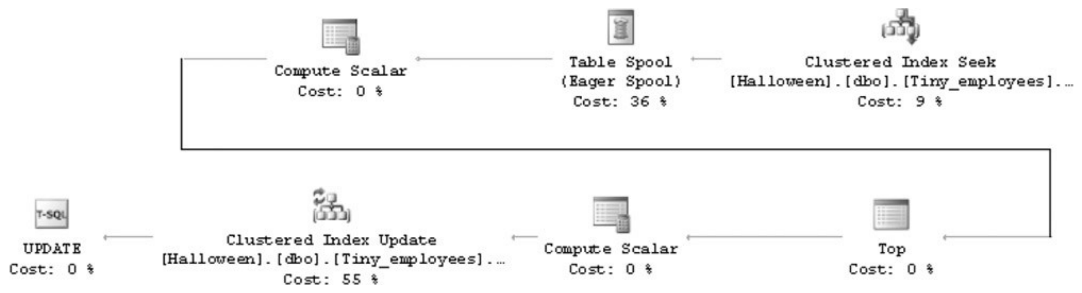


Figure 2-14: Execution plan for the UPDATE statement after creating index ind_salary

```
UPDATE dbo.Tiny_employees
  SET salary = salary * 1.1
WHERE salary < 25000;
```

Once more, there is no update node for the index ind_name because this is a per-row maintenance plan. However, there is a Table Spool operator. I will explain why.

The clustered index seek scans the rows in the order of the clustered key values. We have the clustered index ordered on the *salary* column, and that is the same column updated in our query. Let's assume SQL Server seeks into the clustered index from the smallest value to the largest. The first value encountered in our table is 7500 for the employee emp_E. We increase it by 10 percent to 8250. By that account, the record will be placed between the emp_D and emp_C with "old" salaries of 8000 and 19,000, respectively. Next, we encounter the salary of 8000, and we will increase it to 8800. Figure 2-15 shows the update in progress.

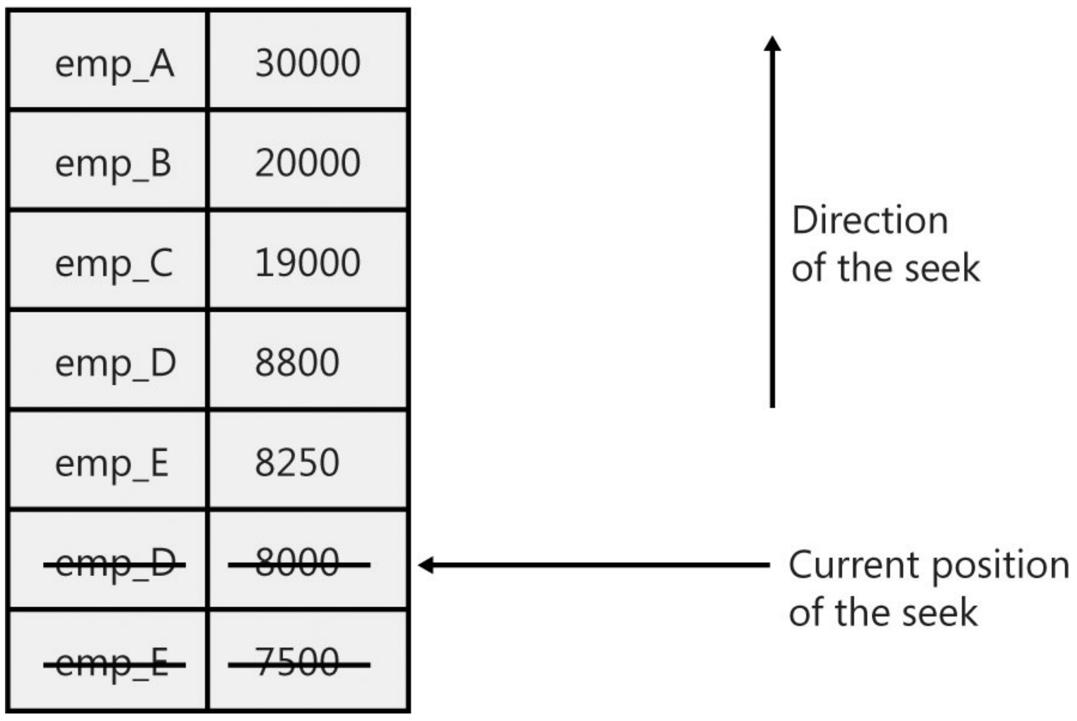


Figure 2-15: Update in progress

If the seek continues, it will reach the emp_E record *again* and update it a second time, and of course, that would be incorrect. Now observe the position of the Table Spool in the preceding query plan—it separates the Clustered Index Seek and Clustered Index Update operators. The spool consumes all records from the index seek before it proceeds with the updates against the same clustered index. The spool, not the index, is producing the updated values. Therefore, the spool prevents updating the same record twice and ensures a correct result. We call such a spool in our IUD query plans a Halloween spool.

You might be wondering what Halloween has in common with the query plans just described? We have to go back almost 30 years in the history of database technology—and query optimization in particular—for the answer. Researchers at the Almaden Research Center in California encountered the same problem when they trained their query optimizer prototype to use indexes in update plans. In fact, they used the *salary* column and salary increase update query similarly to what I just did. Moreover, to their big surprise nobody had a salary under 25,000 after they ran the update query! It was on Halloween 1977 (or maybe 1976?) when the researchers described and debated the glitch. Since then, many database texts, papers, and articles have used the "Halloween" term to name the double-update problem I just outlined.

[*]Thanks goes to Eugene Zabokritski for permission to include information from the patent filed for the algebrizer.

Conclusion

Physical query processing consists of two fundamental steps: query compilation and query execution. The main link between the two steps is the query plan. Understanding how SQL Server generates the query plans and how it is using the plans to deliver the query result is vital for application developers, database designers, and administrators. Constructive use of the knowledge of how SQL Server performs its physical query processing might improve the database server's performance and the application's response time and throughput.

Acknowledgment

I received great help from several colleagues at Microsoft while I was working on this chapter. Mike Blaszczak, Alexis Boukouvalas, Milind Joshi, Stefano Stefani, Don Vilen, Umachandar Jayachandran, and Eugene Zabokritski reviewed and commented on part of the chapter or on the whole chapter and provided me with a wealth of feedback and ideas for improving the content.