Chapter 1

An Introduction to Query Tuning and Optimization

In This Chapter

- Architecture
- Execution Plans
- SET STATISTICS TIME / IO
- Summary
We all have been there: suddenly you get a phone call notifying you of an application outage and asking you to urgently join a conference bridge. After joining the call, you are told that the application is so slow that the company is not able to conduct business; it is losing money and potentially customers too. And usually nobody on the call is able to provide any additional information that can help you find out what the problem is. So, what should you do? Where do you start? And after troubleshooting and fixing the issue, how do you avoid these problems in the future?

Although an outage can be created for several different reasons, including a hardware failure and an operating system problem, as a database professional, you should be able to proactively tune and optimize your databases and be ready to quickly troubleshoot any problem that eventually may occur. This book provides you with the knowledge and tools required to do just that. By focusing on SQL Server performance, and more specifically on query tuning and optimization, this book can help you, first, to avoid these performance problems by optimizing your databases and, second, to quickly troubleshoot and fix them if they happen to appear.

One of the best ways to learn how to improve the performance of your databases is not only to work with the technology, but to understand how the technology works, what it can do for you, how to get the most benefit out of it, and even what its limitations are. The most important SQL Server component impacting the performance of your queries is the SQL Server query processor, which includes the query optimizer and the execution engine. With a perfect query optimizer, you could just submit any query and you would get a perfect execution plan every time. And with a perfect execution engine, each of your queries would run in just a matter of milliseconds. But the reality is that query optimization is a very complex problem, and no query optimizer can find the best plan all the time, at least in a reasonable amount of time. For complex queries, there are so many possible execution plans a query optimizer would need to analyze. And even supposing that a query optimizer could analyze all the possible solutions, the next challenge would be to decide which plan to choose. Which one is the most efficient? Choosing the best plan would require estimating the cost of each solution, which again is a very complicated task.

Don’t get me wrong: the SQL Server query optimizer does an amazing job and gives you a good execution plan almost all the time. But you still need to understand which information you need to provide to the query optimizer so it can do a good job, which may include providing the right indexes and adequate statistics, as well as defining the required constraints and a good database design. SQL Server even provides you with tools to help you in some of these areas, including the Database Engine Tuning Advisor (DTA) and the auto-create and auto-update statistics features. But there is still more you can do to improve the performance of your databases, especially when you are building high-performance applications. Finally, you need to understand the cases where the query optimizer may not give you a good execution plan and what to do in those cases.
Chapter 1: An Introduction to Query Tuning and Optimization

So in order for you to better understand this technology, this chapter starts with an overview of how the SQL Server query processor works and introduces the concepts covered in more detail in the rest of the book. I explain the purpose of both the query optimizer and the execution engine and how they may interact with the plan cache to reuse plans as much as possible. Later, I show you how to work with execution plans, which are the primary tool we’ll use to interact with the query processor.

Architecture

At the core of the SQL Server database engine are two major components: the storage engine and the relational engine, also called the query processor. The storage engine is responsible for reading data between the disk and memory in a manner that optimizes concurrency while maintaining data integrity. The query processor, as the name suggests, accepts all queries submitted to SQL Server, devises a plan for their optimal execution, and then executes the plan and delivers the required results.

Queries are submitted to SQL Server using the SQL language (or T-SQL, the Microsoft SQL Server extension to SQL). Because SQL is a high-level declarative language, it only defines what data to get from the database, not the steps required to retrieve that data or any of the algorithms for processing the request. Thus, for each query it receives, the first job of the query processor is to devise a plan, as quickly as possible, that describes the best possible way (or, at the very least, an efficient way) to execute said query. Its second job is to execute the query according to that plan. Each of these tasks is delegated to a separate component within the query processor; the query optimizer devises the plan and then passes it along to the execution engine, which will actually execute the plan and get the results from the database.

The SQL Server query optimizer is a cost-based optimizer. It analyzes a number of candidate execution plans for a given query, estimates the cost of each of these plans, and selects the plan with the lowest cost of the choices considered. Indeed, given that the query optimizer cannot consider every possible plan for every query, it actually has to find a balance between the optimization time and the quality of the selected plan. Therefore, it is the SQL Server component that has the biggest impact on the performance of your databases. After all, selecting the right or wrong execution plan could mean the difference between a query execution time of milliseconds and one of minutes, or even hours. Naturally, a better understanding of how the query optimizer works can help both database administrators and developers to write better queries and to provide the query optimizer with the information it needs to produce efficient execution plans. This book will demonstrate how you can use your newfound knowledge of the query optimizer’s inner workings; in addition, it will give you the knowledge and tools to troubleshoot the cases when the query optimizer is not giving you a good plan.

In order to arrive at what it believes to be the best plan for executing a query, the query processor performs a number of different steps; the entire query-processing process is shown in Figure 1-1.
We'll look at this whole process in much more detail in Chapter 3, but I'll just run through the steps briefly now:

1. **Parsing and binding** The query is parsed and bound. Assuming the query is valid, the output of this phase is a logical tree, with each node in the tree representing a logical operation that the query must perform, such as reading a particular table or performing an inner join.

2. **Query optimization** The logical tree is then used to run the query optimization process, which roughly consists of the following two steps:

   - **Generation of possible execution plans** Using the logical tree, the query optimizer devises a number of possible ways to execute the query (that is, a number of possible execution plans). An execution plan is, in essence, a set of physical operations (such as an Index Seek or a Nested Loops Join) that can be performed to produce the required result, as described by the logical tree.

   - **Cost assessment of each plan** Although the query optimizer does not generate every possible execution plan, it assesses the resource and time cost of each plan it does generate; the plan that the query optimizer deems to have the lowest cost of those it has assessed is selected and then passed along to the execution engine.
3. **Query execution and plan caching**  The query is executed by the execution engine according to the selected plan; the plan may be stored in memory in the plan cache.

**Parsing and Binding**

Parsing and binding are the first operations performed when a query is submitted to a SQL Server instance. Parsing makes sure that the T-SQL query has a valid syntax, and it translates the SQL query into an initial tree representation: specifically, a tree of logical operators representing the high-level steps required to execute the query in question. Initially, these logical operators will be closely related to the original syntax of the query and will include such logical operations as “get data from the Customer table,” “get data from the Contact table,” “perform an inner join,” and so on. Different tree representations of the query will be used throughout the optimization process, and this logical tree will receive different names until it is finally used to initialize the Memo structure during the optimization process.

Binding is mostly concerned with name resolution. During the binding operation, SQL Server makes sure that all the object names do exist, and it associates every table and column name on the parse tree with its corresponding object in the system catalog. The output of this second process is called an *algebrizer tree*, which is then sent to the query optimizer.

**Query Optimization**

The next step is the optimization process, which is basically the generation of candidate execution plans and the selection of the best of these plans according to their cost. As has already been mentioned, the SQL Server query optimizer uses a cost-estimation model to estimate the cost of each of the candidate plans.

Query optimization could be also seen as the process of mapping the logical query operations expressed in the original tree representation to physical operations, which can be carried out by the execution engine. So, it’s actually the functionality of the execution engine that is being implemented in the execution plans being created by the query optimizer; that is, the execution engine implements a certain number of different algorithms, and it is from these algorithms that the query optimizer must choose when formulating its execution plans. It does this by translating the original logical operations into the physical operations that the execution engine is capable of performing. Execution plans show both the logical and physical operations for each operator. Some logical operations, such as sorts, translate to the same physical operation, whereas other logical operations map to several possible physical operations. For example, a logical join can be mapped to a Nested Loops Join, Merge Join, or Hash Join physical operator. However, this is not a one-to-one operator matching and follows a more complicated process, which will be explained in more detail in Chapter 3.
Thus, the end product of the query optimization process is an execution plan: a tree consisting of a number of physical operators, which contain the algorithms to be performed by the execution engine in order to obtain the desired results from the database.

**Generating Candidate Execution Plans**

As stated, the basic purpose of the query optimizer is to find an efficient execution plan for your query. Even for relatively simple queries, there may be a large number of different ways to access the data to produce the same end result. As such, the query optimizer has to select the best possible plan from what may be a very large number of candidate execution plans, and making a wise choice is important because the time taken to return the results to the user can vary wildly, depending on which plan is selected.

The job of the query optimizer is to create and assess as many candidate execution plans as possible, within certain criteria, in order to find a good enough plan, which may be, but it is not necessarily, the optimal plan. We define the search space for a given query as the set of all possible execution plans for that query, in which any possible plan in this search space returns the same results. Theoretically, in order to find the optimum execution plan for a query, a cost-based query optimizer should generate all possible execution plans that exist in that search space and correctly estimate the cost of each plan. However, some complex queries may have thousands, or even millions, of possible execution plans, and although the SQL Server query optimizer can typically consider a large number of candidate execution plans, it cannot perform an exhaustive search of all the possible plans for every query. If it did, the time taken to assess all of the plans would be unacceptably long and could start to have a major impact on the overall query execution time.

The query optimizer must strike a balance between optimization time and plan quality. For example, if the query optimizer spends one second finding a good enough plan that executes in one minute, then it doesn't make sense to try to find the perfect or most optimal plan if this is going to take five minutes of optimization time, plus the execution time. So SQL Server does not do an exhaustive search, but instead tries to find a suitably efficient plan as quickly as possible. As the query optimizer is working within a time constraint, there's a chance that the plan selected may be the optimal plan, but it is also likely that it may just be something close to the optimal plan.

In order to explore the search space, the query optimizer uses transformation rules and heuristics. The generation of candidate execution plans is performed inside the query optimizer using transformation rules, and the use of heuristics limits the number of choices considered in order to keep the optimization time reasonable. The set of alternative plans considered by the query optimizer is referred to as the plan space, and these plans are stored in memory during the optimization process in a component called the Memo. Transformation rules, heuristics, and the Memo structure will be discussed in more detail in Chapter 3.
Chapter 1: An Introduction to Query Tuning and Optimization

Assessing the Cost of Each Plan

Searching or enumerating candidate plans is just one part of the optimization process. The query optimizer still needs to estimate the cost of these plans and select the least expensive one. To estimate the cost of a plan, it estimates the cost of each physical operator in that plan, using costing formulas that consider the use of resources such as I/O, CPU, and memory. This cost estimation depends mostly on both the algorithm used by the physical operator and the estimated number of records that will need to be processed. This estimate of the number of records is known as the cardinality estimation.

To help with this cardinality estimation, SQL Server uses and maintains statistics, which contain information describing the distribution of values in one or more columns of a table. Once the cost for each operator is estimated using estimations of cardinality and resource demands, the query optimizer will add up all of these costs to estimate the cost for the entire plan. Rather than go into more detail here, I will cover statistics in more detail in Chapter 6.

Query Execution and Plan Caching

Once the query is optimized, the resulting plan is used by the execution engine to retrieve the desired data. The generated execution plan may be stored in memory in the plan cache so it can be reused if the same query is executed again. SQL Server has a pool of memory that is used to store both data pages and execution plans. Most of this memory is used to store database pages, and it is called the buffer pool. A portion of this memory contains the execution plans for queries that were optimized by the query optimizer and is referred to as the plan cache (and was previously known as the procedure cache). The percentage of memory allocated to the plan cache or the buffer pool varies dynamically, depending on the state of the system.

Before optimizing a query, SQL Server first checks the plan cache to see if an execution plan exists for the batch. Query optimization is a relatively expensive operation, so if a valid plan is available in the plan cache, the optimization process can be skipped and the associated cost of this step, in terms of optimization time, CPU resources, and so on, can be avoided. If a plan for the batch is not found, the batch is compiled to generate an execution plan for all queries in the stored procedure, the trigger, or the dynamic SQL batch. Query optimization begins by loading all the interesting statistics. Then the query optimizer validates if the statistics are outdated. For any outdated statistics, when using the statistics default options, it will update the statistics and will proceed with the optimization.

After a plan is found in the plan cache or a new one is created, the plan is validated for schema and data statistics changes. Schema changes are verified for plan correctness. Statistics are also verified: the query optimizer checks for new applicable statistics or outdated statistics. If the plan is not valid for any of these reasons, it is discarded and the batch or individual query is compiled again. Such compilations are known as recompilations. This process is summarized in Figure 1-2.
Plans may also be removed from the plan cache when SQL Server is under memory pressure or when certain statements are executed. Changing some configuration options (for example, max degree of parallelism) will clear the entire plan cache. Alternatively, some statements, such as altering a database with certain ALTER DATABASE options, will clear all the plans associated with that particular database.

However, it is also worth noting that reusing an existing plan may not always be the best solution for a given query, and some problems may appear. For example, depending on the data distribution within a table, the optimal execution plan for a query may differ greatly depending on the parameters being used. More details about these problems and the plan cache in general are covered in Chapter 8.
Chapter 1: An Introduction to Query Tuning and Optimization

Execution Plans

Now that we have a foundation in the query processor and how it works its magic, it’s time to consider how we can interact with it. Primarily, we’ll interact with the query processor through execution plans, which as I mentioned earlier are ultimately trees consisting of a number of physical operators that, in turn, contain the algorithms to produce the required results from the database. Given that I will make extensive use of execution plans throughout the book, in this section, I’ll show you how to display and read them.

You can request either an actual or an estimated execution plan for a given query, and either of these two types can be displayed as a graphic, text, or XML plan. Any of these three formats shows the same execution plan—the only difference is how they are displayed and the level of detail of information they contain.

When an estimated plan is requested, the query is not executed; the plan displayed is simply the plan that SQL Server would most probably use if the query were executed (bearing in mind that a recompile, which we’ll discuss later, may generate a different plan at execution time). However, when an actual plan is requested, the query needs to be executed, and the plan is then displayed along with the query results. Nevertheless, using an estimated plan has several benefits, including displaying a plan for a long-running query for inspection without actually running the query, or displaying a plan for update operations without changing the database.

Graphical Plans

You can display graphical plans in SQL Server Management Studio by clicking the Display Estimated Execution Plan button or the Include Actual Execution Plan button from the SQL Editor toolbar. Clicking Display Estimated Execution Plan will show the plan immediately, without executing the query, whereas, to request an actual execution plan, you need to click Include Actual Execution Plan and then execute the query and click the Execution plan tab.

As an example, copy the following query to the Management Studio Query Editor, select the AdventureWorks2012 database, click the Include Actual Execution Plan button, and execute the query:

```sql
SELECT DISTINCT(City) FROM Person.Address
```

Then select the Execution Plan tab in the results pane. This displays the plan shown in Figure 1-3.
NOTE

This book contains a large number of sample SQL queries, all of which are based on the AdventureWorks2012 database, although Chapter 9 additionally uses the AdventureWorksDW2012 database. All code has been tested on SQL Server 2014 RTM. Note that these sample databases are not included in your SQL Server installation by default, but can be downloaded from the CodePlex website. You need to download the family of sample databases for SQL Server 2012. (No sample databases for SQL Server 2014 existed at the time this book was written.) During installation, you may choose to install all the databases or, at least, AdventureWorks2012 and AdventureWorksDW2012.

Each node in the tree structure is represented as an icon that specifies a logical and physical operator, such as the Index Scan and the Hash Aggregate operators, shown in Figure 1-3. The first icon is a language element called the Result operator, which represents the SELECT statement and is usually the root element in the plan.

Operators implement a basic function or operation of the execution engine; for example, a logical join operation could be implemented by any of three different physical join operators: Nested Loops Join, Merge Join, or Hash Join. Obviously, many more operators are implemented in the execution engine, and you can see the entire list at http://msdn.microsoft.com/en-us/library/ms191158(v=sql.110).aspx. Logical and physical operators’ icons are displayed in blue, except for cursor operators, which are yellow, and language elements are displayed in green:

A logical/physical operator

A language element

A cursor

The query optimizer builds an execution plan, choosing from these operators, which may read records from the database, like the Index Scan operator shown in the previous plan; or they may read records from another operator, like the Hash Aggregate, which reads records from the Index Scan operator.
Chapter 1: An Introduction to Query Tuning and Optimization

Each node is related to a parent node, connected with arrowheads, where data flows from a child operator to a parent operator and the arrow width is proportional to the number of rows. After the operator performs some function on the records it has read, the results are output to its parent. You can hover the mouse pointer over an arrow to get more information about that data flow, displayed as a tooltip. For example, if you hover the mouse pointer over the arrow between the Index Scan and Hash Aggregate operators, shown in Figure 1-3, you get the data flow information between these operators, as shown in Figure 1-4.

By looking at the actual number of rows, you can see that the Index Scan operator is reading 19,614 rows from the database and sending them to the Hash Aggregate operator. The Hash Aggregate operator is, in turn, performing some operation on this data and sending 575 records to its parent, which you can also see by placing the mouse pointer over the arrow between the Hash Aggregate and the SELECT icon.

Basically, in this plan, the Index Scan operator is reading all 19,614 rows from an index, and the Hash Aggregate is processing these rows to obtain the list of distinct cities, of which there are 575, which will be displayed in the Results window in Management Studio. Notice also how you can see the estimated number of rows, which is the query optimizer’s cardinality estimation for this operator, as well as the actual number of rows. Comparing the actual and the estimated number of rows can help you detect cardinality estimation errors, which can affect the quality of your execution plans, as will be discussed in Chapter 6.

To perform their job, physical operators implement at least the following three methods:

- **Open()**  Caused an operator to be initialized, and may include setting up any required data structures
- **GetRow()**  Requests a row from the operator
- **Close()**  Performs some cleanup operations and shuts down the operator once it has performed its role

An operator requests rows from other operators by calling their GetRow() method, which also means that execution in a plan starts from left to right. Because GetRow()
produces just one row at a time, the actual number of rows displayed in the execution plan is also the number of times the method was called on a specific operator, and an additional call to GetRow() is used by the operator to indicate the end of the result set. In the previous example, the Hash Aggregate operator calls the Open() method once, GetRow() 19,615 times, and Close() once on the Index Scan operator.

**NOTE**

For now, I will be explaining the traditional query-processing mode in which operators process only one row at a time. This processing mode has been used in all versions of SQL Server since SQL Server 7.0. Later in Chapter 9, I will touch on the new batch-processing mode, introduced with SQL Server 2012, which is used by operators related to columnstore indexes.

In addition to learning more about the data flow, you can also hover the mouse pointer over an operator to get more information about it. For example, Figure 1-5 shows information about the Index Scan operator; notice that it includes, among other things, a description of the operator and data on estimated costing information, such as the estimated I/O, CPU, operator, and subtree costs.
Chapter 1: An Introduction to Query Tuning and Optimization

Some of these properties are explained in Table 1-1; others will be explained later in the book.

**NOTE**

*It is worth mentioning that the cost is just internal cost units that are not meant to be interpreted in other units such as seconds or milliseconds.*

You can also see the relative cost of each operator in the plan as a percentage of the overall plan, as shown in Figure 1-3. For example, the cost of the Index Scan is 52 percent of the cost of the entire plan. Additional information from an operator or the entire query can be obtained by using the Properties window. So, for example, choosing the SELECT icon and selecting the Properties window from the View menu (or pressing F4) will show some properties for the entire query, as shown in Figure 1-6.

Table 1-2 lists most of the properties shown in Figure 1-6. Other optional properties may appear, depending on the query (for example, Parameter List or Warnings).

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Operation</td>
<td>Physical implementation algorithm for the node.</td>
</tr>
<tr>
<td>Logical Operation</td>
<td>Relational algebraic operator this node represents.</td>
</tr>
<tr>
<td>Actual Number of Rows</td>
<td>Actual number of rows produced by the operator.</td>
</tr>
<tr>
<td>Estimated I/O Cost</td>
<td>Estimated I/O cost for the operation. Not all operators incur an I/O cost.</td>
</tr>
<tr>
<td>Estimated Operator Cost</td>
<td>The query optimizer—estimated cost for executing this operation. This is the estimated I/O and CPU cost. It also includes the cost of operation as a percentage of the total cost of the query displayed in parentheses.</td>
</tr>
<tr>
<td>Estimated Subtree Cost</td>
<td>Estimated cumulative cost for executing this operation and all operations preceding it in the same subtree.</td>
</tr>
<tr>
<td>Estimated CPU Cost</td>
<td>Estimated CPU cost for the operation.</td>
</tr>
<tr>
<td>Estimated Number of Executions</td>
<td>Estimated number of times this operator will be executed while running the current query.</td>
</tr>
<tr>
<td>Number of Executions</td>
<td>Number of times this operator was executed after running the query.</td>
</tr>
<tr>
<td>Estimated Number of Rows</td>
<td>Estimated number of rows produced by the operator (cardinality estimate).</td>
</tr>
<tr>
<td>Estimated Row Size</td>
<td>Estimated average size of the row being passed through this operator.</td>
</tr>
</tbody>
</table>

**Table 1-1  Operator Properties**
### Table 1-2  Query Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cached plan size</td>
<td>Plan cache memory in kilobytes used by this query plan.</td>
</tr>
<tr>
<td>CompileCPU</td>
<td>CPU time in milliseconds used to compile this query.</td>
</tr>
<tr>
<td>CompileMemory</td>
<td>Memory in kilobytes used to compile this query.</td>
</tr>
<tr>
<td>CompileTime</td>
<td>Elapsed time in milliseconds used to compile this query.</td>
</tr>
<tr>
<td>Degree of Parallelism</td>
<td>Number of threads that can be used to execute the query should the query</td>
</tr>
<tr>
<td></td>
<td>processor pick a parallel plan.</td>
</tr>
<tr>
<td>Memory Grant</td>
<td>Amount of memory in kilobytes granted to run this query.</td>
</tr>
<tr>
<td>MemoryGrantInfo</td>
<td>Memory grant estimate as well as actual runtime memory grant information.</td>
</tr>
<tr>
<td>Optimization Level</td>
<td>Level of optimization used to compile this query. Shown as</td>
</tr>
<tr>
<td></td>
<td>StatementOptmLevel on the XML plan. It will be explained in more detail</td>
</tr>
<tr>
<td></td>
<td>later in this section.</td>
</tr>
<tr>
<td>OptimizerHardwareDependentProps</td>
<td>Hardware-dependent properties that affect cost estimate (and hence, query</td>
</tr>
<tr>
<td></td>
<td>plan choice), as seen by the query optimizer.</td>
</tr>
<tr>
<td>QueryHash</td>
<td>Binary hash value calculated on the query and used to identify queries with</td>
</tr>
<tr>
<td></td>
<td>similar logic.</td>
</tr>
<tr>
<td>QueryPlanHash</td>
<td>Binary hash value calculated on the query execution plan and used to</td>
</tr>
<tr>
<td></td>
<td>identify similar query execution plans.</td>
</tr>
<tr>
<td>Reason For Early Termination Of</td>
<td>Shown as StatementOptmEarlyAbortReason on the XML plan. It will be</td>
</tr>
<tr>
<td>Statement Optimization</td>
<td>explained in more detail later in this section.</td>
</tr>
<tr>
<td>RetrievedFromCache</td>
<td>Indicates if plan was retrieved from cache.</td>
</tr>
<tr>
<td>Set Options</td>
<td>Status of the set options that affect query cost. Shown as</td>
</tr>
<tr>
<td></td>
<td>StatementSetOptions on the XML plan. These SET options are ANSI_NULLS,</td>
</tr>
<tr>
<td></td>
<td>ANSI_PADDING, ANSI_WARNINGS, ARITHABORT, CONCAT_NULL_YIELDS_NULL, NUMERIC_</td>
</tr>
<tr>
<td></td>
<td>ROUNDABORT, and QUOTED_IDENTIFIER.</td>
</tr>
<tr>
<td>Statement</td>
<td>Text of the SQL statement.</td>
</tr>
</tbody>
</table>

Finally, SQL Server Management Studio provides a zooming feature you can use to navigate through large graphical plans that may not fit on the screen. You can access this tool by clicking the plus sign button located at the bottom-right corner of the Execution Plan tab. An example is shown in Figure 1-7. In addition, SQL Sentry Plan Explorer is a popular tool you can use to work with execution plans. You can download it for free at the SQL Sentry website at http://sqlsentry.net/plan-explorer.
Figure 1-6  Properties window for the query

Figure 1-7  Execution plans zooming feature
XML

Once you have displayed a graphical plan, you can also easily display it in XML format. Simply right-click anywhere on the execution plan window to display a pop-up menu, as shown in Figure 1-8, and select Show Execution Plan XML…. This will open the XML editor and display the XML plan. As you can see, you can easily switch between a graphical and an XML plan.

If needed, you can save graphical plans to a file by selecting Save Execution Plan As… from the pop-up menu shown in Figure 1-8. The plan, usually saved with an .sqlplan extension, is actually an XML document containing the XML plan, but can be read by Management Studio into a graphical plan. You can load this file again by selecting File and Open in Management Studio in order to immediately display it as a graphical plan, which will behave exactly as before. XML plans can also be used with the USEPLAN query hint, which is explained in Chapter 10.

Table 1-3 shows the different statements you can use to obtain an estimated or actual execution plan in text, graphic, or XML format.

NOTE

When you run any of the statements listed in Table 1-3 using the ON clause, it will apply to all subsequent statements until the option is manually set to OFF again.

To show an XML plan, you can use the following commands:

```
SET SHOWPLAN_XML ON
GO
SELECT DISTINCT(City) FROM Person.Address
GO
SET SHOWPLAN_XML OFF
```

![Figure 1-8 Pop-up menu on the execution plan window](image-url)
Chapter 1: An Introduction to Query Tuning and Optimization

---

This will display a single-row, single-column (titled “Microsoft SQL Server 2005 XML Showplan”) result set containing the XML data that starts with the following:

```xml
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004 ...
```

Clicking the link will show you a graphical plan, and you can then display the XML plan using the same procedure as explained earlier.

You can browse the basic structure of an XML plan via the following exercise. A very simple query will create the basic XML structure, but in this example I show you a query that can provide two additional parts: the missing indexes and parameter list elements. Run the following query and request an XML plan:

```sql
SELECT * FROM Sales.SalesOrderDetail
WHERE OrderQty = 1
```

Collapse `<MissingIndexes>`, `<RelOp>`, and `<ParameterList>` by clicking the minus sign (-) on the left so you can easily see the entire structure. You should see something similar to Figure 1-9.

---

### Table 1-3 Statements for Displaying Query Plans

<table>
<thead>
<tr>
<th></th>
<th>Estimated Execution Plan</th>
<th>Actual Execution Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Plan</td>
<td>SET SHOWPLANTEXT</td>
<td>SET STATISTICS PROFILE</td>
</tr>
<tr>
<td></td>
<td>SET SHOWPLANALL</td>
<td></td>
</tr>
<tr>
<td>Graphic Plan</td>
<td>Management Studio</td>
<td>Management Studio</td>
</tr>
<tr>
<td>XML Plan</td>
<td>SET SHOWPLANXML</td>
<td>SET STATISTICS XML</td>
</tr>
</tbody>
</table>

---

*Figure 1-9 XML execution plan*
As you can see, the main components of the XML plan are the <StmtSimple>, <StatementSetOptions>, and <QueryPlan> elements. These three elements include several attributes, some of which were already explained when we discussed the graphical plan. In addition, the <QueryPlan> element also includes other elements such as <MissingIndexes>, <MemoryGrantInfo>, <OptimizerHardwareDependentProperties>, <RelOp>, <ParameterList>, and others not shown in Figure 1-9, such as <Warnings>, which will be also discussed later in this section. <StmtSimple> shows the following for this example:

```
<StmtSimple StatementCompId="1" StatementEstRows="68089" StatementId="1" StatementOptmLevel="FULL" CardinalityEstimationModelVersion="70" StatementSubTreeCost="1.13478" StatementText="SELECT * FROM [Sales].[SalesOrderDetail] WHERE [OrderQty]=@1" StatementType="SELECT" QueryHash="0x42CFD97ABC9592DD" QueryPlanHash="0xC5F6C30459CD7C41" RetrievedFromCache="false">

<QueryPlan> shows this:

```

```
<QueryPlan DegreeOfParallelism="1" CachedPlanSize="32" CompileTime="3" CompileCPU="3" CompileMemory="264">

As mentioned, the attributes of these and other elements were already explained when we discussed the graphical plan. Others will be explained later in this section or in other sections of the book.

**Text Plans**

As you can see in Table 1-3, there are two commands to get estimated text plans: SET SHOWPLAN_TEXT and SET SHOWPLAN_ALL. Both statements show the estimated execution plan, but SET SHOWPLAN_ALL shows some additional information, including the estimated number of rows, estimated CPU cost, estimated I/O cost, and estimated operator cost. However, recent versions of Books Online, including that of SQL Server 2014, indicate that all text versions of execution plans will be deprecated in a future version of SQL Server and, therefore, recommend using the XML versions instead.

You can use the following code to display a text execution plan:

```
SET SHOWPLAN_TEXT ON
GO
SELECT DISTINCT(City) FROM Person.Address
GO
SET SHOWPLAN_TEXT OFF
GO
```
Chapter 1: An Introduction to Query Tuning and Optimization

This code will actually display two result sets, the first one returning the text of the T-SQL statement. In the second result set, you see the following text plan (edited to fit the page), which shows the same Hash Aggregate and Index Scan operators displayed earlier in Figure 1-3:

```
|--Hash Match(Aggregate, HASH:([Person].[Address].[City]), RESIDUAL ...
|--Index Scan(OBJECT:([AdventureWorks].[Person].[Address].[IX_Address ...
```

SET SHOWPLAN_ALL and SET STATISTICS PROFILE can provide more detailed information than SET SHOWPLAN_TEXT. Also, as shown in Table 1-3, you can use SET SHOWPLAN_ALL to get an estimated plan only and SET STATISTICS PROFILE to actually execute the query. Run the following example:

```
SET SHOWPLAN_ALL ON
GO
SELECT DISTINCT(City) FROM Person.Address
GO
SET SHOWPLAN_ALL OFF
GO
```

The output is shown in Figure 1-10.

Because SET STATISTICS PROFILE actually executes the query, it provides an easy way to look for cardinality estimation problems because you can easily visually compare multiple operators at a time, which could be complicated to do on a graphical or XML plan. Now run the following code:

```
SET STATISTICS PROFILE ON
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE OrderQty * UnitPrice > 25000
GO
SET STATISTICS PROFILE OFF
GO
```

The output is shown in Figure 1-11.

Note that the column EstimateRows was manually moved in Management Studio to be next to the column Rows so you can easily compare the actual against the estimated number of rows. For this particular example, you can see a big difference in cardinality estimation on the Filter operator of 36,395.1 estimated versus five actual rows.

---

**Figure 1-10**  SET SHOWPLAN_ALL output

---

<table>
<thead>
<tr>
<th>Step</th>
<th>Text</th>
<th>StrId</th>
<th>UsedId</th>
<th>Parent</th>
<th>Physicup</th>
<th>Logicalup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT DISTINCT(City) FROM Person.Address</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>Hash Match</td>
<td>Aggregate HASH:([AdventureWorks...</td>
<td>1</td>
<td>2</td>
<td>1 Hash Match Aggregate</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Index Scan(OBJECT:([AdventureWorks...</td>
<td>1</td>
<td>3</td>
<td>2 Index Scan Index Scan</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
One interesting way to learn about the components of an execution plan, including ones of future versions of SQL Server, is to look at the showplan schema. XML plans comply with a published XSD schema, and you can see the current and older versions of this showplan schema at http://schemas.microsoft.com/sqlserver/2004/07/showplan/, which you can also find at the beginning of each XML execution plan. Currently, accessing that location on a web browser will show you links to access the showplan schemas for SQL Server 2014 RTM as “Current version,” SQL Server 2012 RTM, SQL Server 2008 RTM, SQL Server 2005 SP2, and SQL Server 2005 RTM.

Covering all the elements and attributes of an execution plan would take a lot of pages; instead, I cover some of the most interesting ones here. Operators used in execution plans will be covered in more detail in Chapter 4. Let’s start with the StatementOptmLevel, StatementOptmEarlyAbortReason, and CardinalityEstimationModelVersion attributes of the <StmtSimple> element.

Although these attributes refer to concepts that will be explained in more detail later in the book, it’s worth an introduction here. StatementOptmLevel is the query optimization level, which can be either TRIVIAL or FULL. The optimization process may be expensive to initialize and run for very simple queries that don’t require any cost estimation, so to avoid this expensive operation for these simple queries, SQL Server uses the trivial plan optimization. If a query does not apply for a trivial optimization, a full optimization will have to be performed. For example, in SQL Server 2014, the following query will produce a trivial plan:

```
SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43666
```

You can use the undocumented (and therefore unsupported) trace flag 8757 to test the behavior if you want to disable the trivial plan optimization.

```
SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43666
OPTION (QUERYTRACEON 8757)
```
The QUERYTRACEON query hint is used to apply a trace flag at the query level. After running the previous query, SQL Server will run a full optimization, which you can verify with StatementOptmLevel as FULL in the resulting plan. You should note that although the QUERYTRACEON query hint is widely known, currently, it is only supported in a limited number of scenarios. At the time of this writing, the QUERYTRACEON query hint is only supported when using the trace flags documented in the article found at http://support.microsoft.com/kb/2801413.

Note
This book shows many undocumented and unsupported features. This is so you can use them in a test environment for troubleshooting purposes or to learn the technology, but they are not meant to be used in a production environment and are not supported by Microsoft. I will identify when a statement or trace flag is undocumented and unsupported.

On the other hand, the StatementOptmEarlyAbortReason, or “Reason For Early Termination Of Statement Optimization,” attribute can have the values GoodEnoughPlanFound, TimeOut, and MemoryLimitExceeded and only appears when the query optimizer prematurely terminates a query optimization (in older versions of SQL Server, you had to use undocumented trace flag 8675 to see this information). Because the purpose of the query optimizer is to produce a good enough plan as quickly as possible, the query optimizer calculates two values depending on the query at the beginning of the optimization process. The first of these values is the cost of a good enough plan according to the query, and the second one is the maximum time to spend on the query optimization. During the optimization process, if a plan with a cost lower than the calculated cost threshold is found, the optimization process stops, and the found plan will be returned with the GoodEnoughPlanFound value.

If, on the other hand, the optimization process is taking longer than the calculated maximum time threshold, optimization will also stop and the query optimizer will return the best plan found so far, with StatementOptmEarlyAbortReason containing the TimeOut value. The GoodEnoughPlanFound and TimeOut values do not mean that there is a problem, and in all three cases, including MemoryLimitExceeded, the plan produced will be correct. However, in the case of MemoryLimitExceeded, the plan may not be optimal. In this latter case, you may need to simplify your query or increase the available memory in your system. These and other details of the query optimization process are covered in Chapter 3.

For example, even when the following query joins four tables and requires a Sort, it still has an early termination and returns “Good Enough Plan Found”:

```
SELECT pm.ProductModelID, pm.Name, Description, pl.CultureID, cl.Name AS Language
FROM Production.ProductModel AS pm
    JOIN Production.ProductModelProductDescriptionCulture AS pl
```
ON pm.ProductModelID = pl.ProductModelID
JOIN Production.Culture AS cl
ON cl.CultureID = pl.CultureID
JOIN Production.ProductDescription AS pd
ON pd.ProductDescriptionID = pl.ProductDescriptionID
ORDER BY pm.ProductModelID

The CardinalityEstimationModelVersion attribute refers to the version of the cardinality estimation model used by the query optimizer. SQL Server 2014 introduces a new cardinality estimator, but you still have the choice of using the old one by changing the database compatibility level or using trace flags 2312 and 9481. More details about both cardinality estimation models will be covered in Chapter 6.

The NonParallelPlanReason optional attribute of the QueryPlan element, which was introduced with SQL Server 2012, contains a description of why a parallel plan may not be chosen for the optimized query. Although the list of possible values is not documented, the following are popular and easy to obtain:

SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43666
OPTION (MAXDOP 1)

Because we are using MAXDOP 1, it will show this:

NonParallelPlanReason="MaxDOPSetToOne"

Using the function

SELECT CustomerID, ('AW' + dbo.ufnLeadingZeros(CustomerID))
AS GenerateAccountNumber
FROM Sales.Customer
ORDER BY CustomerID;

would generate the following:

NonParallelPlanReason="CouldNotGenerateValidParallelPlan"

If trying to run

SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 43666
OPTION (MAXDOP 8)

on a system with only one CPU, you will get this:

<QueryPlan NonParallelPlanReason="EstimatedDOPIsOne"
Finally, also introduced with SQL Server 2012, the showplan XSD schema has the OptimizerHardwareDependentProperties element, which provides hardware-dependent properties that can affect the query plan choice, with the following documented attributes:

- **EstimatedAvailableMemoryGrant** An estimate of what amount of memory (KB) will be available for this query at execution time to request a memory grant from
- **EstimatedPagesCached** An estimate of how many pages of data will remain cached in the buffer pool if the query needs to read it again
- **EstimatedAvailableDegreeOfParallelism** An estimate of the number of CPUs that can be used to execute the query should the query optimizer pick a parallel plan

For example, the query

```sql
SELECT DISTINCT(CustomerID)
FROM Sales.SalesOrderHeader
```

will show this:

```xml
<OptimizerHardwareDependentProperties
  EstimatedAvailableMemoryGrant="101808"
  EstimatedPagesCached="8877"
  EstimatedAvailableDegreeOfParallelism="2" />
```

### Warnings on Execution Plans

Execution plans can also show warning messages. Plans containing these warnings should be carefully reviewed because this can cause the query optimizer to choose a less efficient query plan. Before SQL Server 2012, only the ColumnsWithNoStatistics and NoJoinPredicate were available. The SQL Server 2012 showplan schema added six more iterator- or query-specific warnings:

- **SpillToTempDb**
- **Wait**
- **PlanAffectingConvert**
- **SpatialGuess**
- **UnmatchedIndexes**
- **FullUpdateForOnlineIndexBuild**

Let’s examine some of them in this section.
**ColumnsWithNoStatistics**

This warning means that the query optimizer tried to use statistics but none were available. As explained earlier in this chapter, the query optimizer relies on statistics to produce an optimal plan. Perform the following statements to simulate this warning.

Run the following statement to drop the existing statistics for the VacationHours column, if available:

```sql
DROP STATISTICS HumanResources.Employee._WA_Sys_0000000C_49C3F6B7
```

Next, temporarily disable automatic creation of statistics at the database level:

```sql
ALTER DATABASE AdventureWorks2012 SET AUTO_CREATE_STATISTICS OFF
```

Then run this query:

```sql
SELECT * FROM HumanResources.Employee
WHERE VacationHours = 48
```

You will get the plan shown in Figure 1-12.

Notice the warning (the symbol with an exclamation mark) on the Clustered Index Scan operator. If you look at its properties you will see “Columns With No Statistics: [AdventureWorks2012].[HumanResources].[Employee].VacationHours.”

Do not forget to reenable the automatic creation of statistics by running the following command. There is no need to create the statistics object dropped previously because it can be created automatically if needed.

```sql
ALTER DATABASE AdventureWorks2012 SET AUTO_CREATE_STATISTICS ON
```

**NoJoinPredicate**

A possible problem while using the old-style ANSI SQL-89 join syntax is accidentally missing the join predicate and getting a NoJoinPredicate warning. Let’s suppose you intend to run the following query but forgot to include the WHERE clause:

```sql
SELECT * FROM Sales.SalesOrderHeader soh, Sales.SalesOrderDetail sod
WHERE soh.SalesOrderID = sod.SalesOrderID
```

The first indication of a problem could be that the query takes way too long to execute, even for small tables. Later, you will see that the query also returns a huge result set.
Chapter 1: An Introduction to Query Tuning and Optimization

Sometimes, a way to troubleshoot a long-running query is to just stop its execution and request an estimated plan instead. If you don’t include the join predicate (in the WHERE clause), you get the plan shown in Figure 1-13.

This time, you can see the warning on the Nested Loops Join as “No Join Predicate” with a different symbol. Notice that you cannot accidentally miss a join predicate if you use the ANSI SQL-92 join syntax because you get an error instead, which is why this syntax is recommended. For example, missing the join predicate in the following query will return an incorrect syntax error:

```sql
SELECT * FROM Sales.SalesOrderHeader soh JOIN Sales.SalesOrderDetail sod
-- ON soh.SalesOrderID = sod.SalesOrderID
```

**NOTE**

You can still get, if needed, a join whose result set includes one row for each possible pairing of rows from the two tables, also called a Cartesian product, by using the CROSS JOIN syntax.

**PlanAffectingConvert**

This warning shows that type conversions were performed that may impact the performance of the resulting execution plan. Run the following example, which declares a variable as nvarchar and then uses it in a query to compare against a varchar column, CreditCardApprovalCode:

```sql
DECLARE @code nvarchar(15)
SET @code = '95555Vi4081'
SELECT * FROM Sales.SalesOrderHeader
WHERE CreditCardApprovalCode = @code
```

The query returns the plan shown in Figure 1-14.
The following two warnings are shown on the SELECT icon:

Type conversion in expression
(CONVERT_IMPLICIT(nvarchar(15), [AdventureWorks2012].[Sales].[SalesOrderHeader].
[CreditCardApprovalCode], 0)) may affect "CardinalityEstimate" in query plan choice,

Type conversion in expression
(CONVERT_IMPLICIT(nvarchar(15), [AdventureWorks2012].[Sales].[SalesOrderHeader].
[CreditCardApprovalCode], 0) = [@code]) may affect "SeekPlan" in query plan choice

Obviously, the recommendation is to use similar data types on comparison operations.

**SpillToTempDb**

This warning shows than an operation didn't have enough memory and had to spill data
to disk during execution, which can be a performance problem because of the extra I/O
overhead. To simulate this problem, run the following example:

SELECT * FROM Sales.SalesOrderDetail
ORDER BY UnitPrice

This is a very simple query, and depending on the memory available on your system,
you may not get the warning in your test environment, so you may need to try with a
larger table instead. The plan shown in Figure 1-15 is generated.

The warning is shown this time on the Sort operator, which includes the message
“Operator used tempdb to spill data during execution with spill level 1.” The XML
plan also shows this:

<SpillToTempDb SpillLevel="1" />

Figure 1-15  Plan with SpillToTempDb warning
Chapter 1: An Introduction to Query Tuning and Optimization

UnmatchedIndexes

Finally, the UnmatchedIndexes element can show that the query optimizer was not able to match a filtered index for a particular query (for example, when it is not able to see the value of a parameter). Suppose you create the following filtered index:

```
CREATE INDEX IX_Color ON Production.Product(Name, ProductNumber)
WHERE Color = 'White'
```

Then you run the following query:

```
DECLARE @color nvarchar(15)
SET @color = 'White'
SELECT Name, ProductNumber FROM Production.Product
WHERE Color = @color
```

The IX_Color index is not used at all, and you will get a warning on the plan, as shown in Figure 1-16.

You will be able to see the following on the XML plan (or by looking at the UnmatchedIndexes property of the SELECT operator properties window):

```
<UnmatchedIndexes>
  <Parameterization>
    <Object Database="[AdventureWorks2012]" Schema="[Production]"
      Table="[Product]" Index="[IX_Color]" />
  </Parameterization>
</UnmatchedIndexes>
``` 

However, the following query will use the index:

```
SELECT Name, ProductNumber FROM Production.Product
WHERE Color = 'White'
```

Filtered indexes and the UnmatchedIndexes element will be covered in detail in Chapter 5. For now, remove the index we just created:

```
DROP INDEX Production.Product.IX_Color
```

Figure 1-16 Plan with a UnmatchedIndexes warning
NOTE
Many exercises in this book will require you to perform changes in the AdventureWorks2012 database. Although the database is reverted back to its original state, you may also consider refreshing a copy of the database after a number of changes or tests.

Getting Plans from a Trace or the Plan Cache

So far we have been testing getting execution plans by directly using the query code in Management Studio. However, getting a plan this way may not always be possible in your environment, and sometimes, you may need to capture, perhaps for troubleshooting purposes, the execution plan from other locations (for example, the plan cache or a current execution). In these cases, you may need to obtain an execution plan from a trace, for example, using SQL trace or extended events, or the plan cache using the sys.dm_exec_query_plan dynamic management function (DMF) or perhaps using some collected data, as in the case of the SQL Server Data Collector. Let’s take a look at some of these sources.

sys.dm_exec_query_plan DMF

As mentioned earlier, when a query is optimized, its execution plan may be kept in the plan cache, and the sys.dm_exec_query_plan DMF can be used to return such cached plans, as well as any plan that is currently executing. However, when a plan is removed from the cache, it will no longer be available and the query_plan column of the returned table will be null.

For example, the following query shows the execution plans for all the queries currently running in the system. The sys.dm_exec_requests dynamic management view (DMV), which returns information about each request currently executing, is used to obtain the plan_handle value, which is needed to find the execution plan using the sys.dm_exec_query_plan DMF. A plan_handle is a hash value that represents a specific execution plan, and it is guaranteed to be unique in the system.

```
SELECT * FROM sys.dm_exec_requests
CROSS APPLY
sys.dm_exec_query_plan(plan_handle)
```

The output will be a result set containing the query_plan column, which shows links similar to the one shown in the XML plans section. As explained before, clicking the link shows you requested the graphical execution plan.

In the same way, the following example shows the execution plans for all cached query plans. The sys.dm_exec_query_stats DMV contains one row per query statement.
within the cached plan and, again, provides the plan_handle value needed by the sys.dm_exec_query_plan DMF.

```
SELECT * FROM sys.dm_exec_query_stats
CROSS APPLY
sys.dm_exec_query_plan(plan_handle)
```

Now suppose you want to find the 10 most expensive queries by CPU usage. You can run the following query to get this information, which will return the average CPU time in microseconds per execution:

```
SELECT TOP 10 total_worker_time/execution_count AS avg_cpu_time,
    plan_handle, query_plan
FROM sys.dm_exec_query_stats
CROSS APPLY sys.dm_exec_query_plan(plan_handle)
ORDER BY avg_cpu_time DESC
```

**SQL Trace/Profiler**

You can also use SQL Trace and/or Profiler to capture execution plans of queries currently executing. You can use the Performance event category in Profiler, which includes the following events:

- Performance Statistics
- Showplan All
- Showplan All For Query Compile
- Showplan Statistics Profile
- Showplan Text
- Showplan Text (Unencoded)
- Showplan XML
- Showplan XML For Query Compile
- Showplan XML Statistics Profile

To trace any of these events, run Profiler, connect to your SQL Server instance, click Events Selection, expand the Performance event category, and select any of the required events. You can select all the columns or only a subset of the columns, specify a column filter, and so on. Click Run to start the trace. Figure 1-17 shows an example of a trace with the Showplan XML event.
You can optionally create a server trace using a script, and even use Profiler as a scripting tool. In order to do that, define the events for your trace, run and stop the trace, and select File | Export | Script Trace Definition | For SQL Server 2005 – 2014.... This will produce the code to run a server trace, which will only be required to specify a filename to capture the trace. Part of the generated code is shown next.

```sql
-- Create a Queue
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

-- Please replace the text InsertFileNameHere, with an appropriate
-- filename prefixed by a path, e.g., c:\MyFolder\MyTrace. The .trc extension
-- will be appended to the filename automatically. If you are writing from
-- remote server to local drive, please use UNC path and make sure server has
-- write access to your network share
```

**Figure 1-17  Trace in Profiler showing the Showplan XML event**
Chapter 1: An Introduction to Query Tuning and Optimization

exec @rc = sp_trace_create @TraceID output, 0, N'InsertFileNameHere', @maxfilesize, NULL
if (@rc != 0) goto error

-- Client side File and Table cannot be scripted

-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 10, 1, @on
exec sp_trace_setevent @TraceID, 10, 9, @on
exec sp_trace_setevent @TraceID, 10, 2, @on
exec sp_trace_setevent @TraceID, 10, 66, @on
exec sp_trace_setevent @TraceID, 10, 10, @on
exec sp_trace_setevent @TraceID, 10, 3, @on
exec sp_trace_setevent @TraceID, 10, 4, @on
exec sp_trace_setevent @TraceID, 10, 6, @on
exec sp_trace_setevent @TraceID, 10, 7, @on
exec sp_trace_setevent @TraceID, 10, 8, @on
exec sp_trace_setevent @TraceID, 10, 11, @on
exec sp_trace_setevent @TraceID, 10, 12, @on
exec sp_trace_setevent @TraceID, 10, 13, @on

NOTE

As of SQL Server 2008, all non-XML events mentioned earlier, such as Showplan All, Showplan Text, and so on, are deprecated. Microsoft recommends using the XML events instead. SQL Trace has also been deprecated as of SQL Server 2012. Instead, Microsoft recommends using extended events.

For more details about using Profiler and SQL Trace, refer to SQL Server Books Online.

Extended Events

You can also use extended events to capture execution plans. Although in general, Microsoft recommends using extended events over SQL Trace, as mentioned earlier, the events to capture execution plans are expensive to collect on the current releases of SQL Server. The documentation shows the following warning for all three extended events available to capture execution plans: “Using this event can have a significant performance overhead so it should only be used when troubleshooting or monitoring specific problems for brief periods of time.”
You can create and start an extended events session by using CREATE EVENT SESSION and ALTER EVENT SESSION. You can also use the new graphic user interface introduced in SQL Server 2012. Here are the events related to execution plans:

- **query_post_compilation_showplan** Occurs after a SQL statement is compiled. This event returns an XML representation of the estimated query plan that is generated when the query is compiled.
- **query_post_execution_showplan** Occurs after a SQL statement is executed. This event returns an XML representation of the actual query plan.
- **query_pre_execution_showplan** Occurs after a SQL statement is compiled. This event returns an XML representation of the estimated query plan that is generated when the query is optimized.

For example, let's suppose you want to start a session to trace the query_post_execution_showplan event. You could use the following code to create the extended event session:

```sql
CREATE EVENT SESSION [test] ON SERVER
ADD EVENT sqlserver.query_post_execution_showplan(
    ACTION(sqlserver.plan_handle)
    WHERE ([sqlserver].[database_name]=N'AdventureWorks2012'))
ADD TARGET package0.ring_buffer
WITH (STARTUP_STATE=OFF)
GO
```

More details about extended events will be covered in Chapter 2. In the meantime, you can notice that the ADD EVENT argument shows the event name (in this case, query_post_execution_showplan), ACTION refers to global fields you want to capture in the event session (in this case, plan_handle), and WHERE is used to apply a filter to limit the data you want to capture. The predicate [sqlserver].[database_name]=N'AdventureWorks2012' indicates that we want to capture events for the AdventureWorks2012 database only. TARGET is the event consumer, and we can use it to collect the data for analysis. In this case, we are using the ring buffer target. Finally, STARTUP_STATE is one of the extended event options, and it is used to specify whether or not this event session is automatically started when SQL Server starts.

After the event session is created, you can start it using the ALTER EVENT SESSION statement, as in the following example:

```sql
ALTER EVENT SESSION [test]
ON SERVER
STATE=START
```
You can use the Watch Live Data feature, introduced with SQL Server 2012, to view the data captured by the event session. In order to do that, expand the Management folder in Object Explorer | Extended Events | Sessions, right-click the extended event session, and select Watch Live Data. Figure 1-18 shows how an example of an execution plan is captured.

You can also run the following code to see this data:

```
SELECT event_data.value('(@name)[1]', 'varchar(50)') AS event_name,
       event_data.value('(@name="plan_handle"/value)[1]',
                        'varchar(max)') AS plan_handle,
       event_data.query('event/data[@name="showplan_xml"]/value/*') as showplan_xml,
       event_data.value('(@name="sql_text"/value)[1]',
                        'varchar(max)') AS sql_text
FROM (SELECT evnt.query('.') AS event_data
      FROM (SELECT CAST(target_data AS xml) AS target_data
              FROM sys.dm_xe_sessions AS s
              JOIN sys.dm_xe_session_targets AS t
              ON s.address = t.event_session_address
              WHERE s.name = 'test'
              AND t.target_name = 'ring_buffer'
          ) AS data
      CROSS APPLY target_data.nodes('RingBufferTarget/event') AS xevent(evnt)
     ) AS xevent(event_data)
```

Figure 1-18  Watch Live Data feature
After you finish your test, you need to stop and delete the event session. Run the following statements:

```sql
ALTER EVENT SESSION [test] ON SERVER
STATE=STOP
GO
DROP EVENT SESSION [test] ON SERVER
```

Finally, some other SQL Server tools can allow you to see plans, including the Data Collector. The Data Collector was introduced with SQL Server 2008 and will be covered in Chapter 2.

### Removing Plans from the Plan Cache

You can use a few different commands to remove plans from the plan cache. These commands, covered in more detail in Chapter 8, can be useful during your testing and should not be executed in a production environment, unless the requested effect is desired. The `DBCC FREEPROCCACHE` statement can be used to remove all the entries from the plan cache. It can also accept a plan handle or a SQL handle to remove only specific plans, or a Resource Governor pool name to remove all the cache entries associated with it. The `DBCC FREESYSTEMCACHE` statement can be used to remove all the elements from the plan cache or only the elements associated with a Resource Governor pool name. `DBCC FLUSHPROCINDB` can be used to remove all the cached plans for a particular database.

Finally, although not related to the plan cache, the `DBCC DROPCLEANBUFFERS` statement can be used to remove all the buffers from the buffer pool. You can use this statement in cases where you want to simulate a query starting with a cold cache, as we will do in the next section.

### SET STATISTICS TIME / IO

We close this chapter with two statements that can give you additional information about your queries and that you can use as an additional tuning technique. These can be a great complement to execution plans to get additional information about your queries’ optimization and execution. One common misunderstanding I see is developers trying to compare plan cost to plan performance. You should not assume a direct correlation between a query-estimated cost and its actual runtime performance. Cost is an internal unit used by the query optimizer and should not be used to compare plan performance; `SET STATISTICS TIME` and `SET STATISTICS IO` can be used instead. This section explains both statements.
Chapter 1: An Introduction to Query Tuning and Optimization

You can use SET STATISTICS TIME to see the number of milliseconds required to parse, compile, and execute each statement. For example, run

```
SET STATISTICS TIME ON
```

and then run the following query:

```
SELECT DISTINCT(CustomerID)
FROM Sales.SalesOrderHeader
```

To see the output, you will have to look at the Messages tab of the edit window, which will show an output similar to the following:

```
SQL Server parse and compile time:
   CPU time = 16 ms, elapsed time = 226 ms.

SQL Server Execution Times:
   CPU time = 16 ms, elapsed time = 148 ms.
```

“Parse and compile” refers to the time SQL Server takes to optimize the SQL statement, as explained earlier. SET STATISTICS TIME will continue to be enabled for any subsequently executed queries. You can disable it like so:

```
SET STATISTICS TIME OFF
```

As mentioned previously, parse and compile information can also be seen on the execution plan, as in the following:

```
<QueryPlan DegreeOfParallelism="1" CachedPlanSize="16" CompileTime="226"
CompileCPU="9" CompileMemory="232">
```

Obviously, if you only need the execution time of each query, you can see this information in the Management Studio Query Editor.

SET STATISTICS IO displays the amount of disk activity generated by a query. To enable it, run the following statement:

```
SET STATISTICS IO ON
```

Run this next statement to clean all the buffers from the buffer pool to make sure that no pages for this table are loaded in memory:

```
DBCC DROPCLEANBUFFERS
```
Then run the following query:

```sql
SELECT * FROM Sales.SalesOrderDetail
WHERE ProductID = 870
```

It will show an output similar to the following:

Table 'SalesOrderDetail'. Scan count 1, logical reads 1246, physical reads 3,
read-ahead reads 1277, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.

Here are the definitions of these items, which all use 8K pages:

- **Logical reads**  Number of pages read from the buffer pool.
- **Physical reads**  Number of pages read from disk.
- **Read-ahead reads**  Read-ahead is a performance optimization mechanism that
  anticipates the needed data pages and reads them from disk. It can read up to 64
  contiguous pages from one data file.
- **Lob logical reads**  Number of large object (LOB) pages read from the buffer pool.
- **Lob physical reads**  Number of large object (LOB) pages read from disk.
- **Lob read-ahead reads**  Number of large object (LOB) pages read from disk using
  the read-ahead mechanism, as explained earlier.

Now, if you run the same query again, you will no longer get physical and read-ahead
reads, and you will get an output similar to this:

Table 'SalesOrderDetail'. Scan count 1, logical reads 1246, physical reads 0,
read-ahead reads 0, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.

“Scan count” is defined as the number of seeks or scans started after reaching the leaf
level (that is, the bottom level of an index). The only case when scan count will return
0 is when you’re seeking for only one value on a unique index, like in the following
example:

```sql
SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID = 51119
```
If you try the following query, in which SalesOrderID is defined in a nonunique index and can return more than one record, you can see that scan count now returns 1:

```sql
SELECT * FROM Sales.SalesOrderDetail
WHERE SalesOrderID = 51119
```

Finally, in the following example, scan count is 4 because SQL Server has to perform four seeks:

```sql
SELECT * FROM Sales.SalesOrderHeader
WHERE SalesOrderID IN (51119, 43664, 63371, 75119)
```

**Summary**

In this chapter, I showed you how a better understanding of what the query processor does behind the scenes can help both database administrators and developers write better queries and provide the query optimizer with the information it needs to produce efficient execution plans. In the same way, I showed you how you can use your newfound knowledge of the query processor inner workings and SQL Server tools to troubleshoot the cases when your queries are not performing as expected. Based on this, the basics of the query optimizer, the execution engine, and the plan cache were explained, and these SQL Server components will be covered in greater detail in later chapters.

Because we will be using execution plans throughout the entire book, I also introduced you to how to read them, their more important properties, and how to obtain them from sources such as the plan cache and a server trace. This section should have given you enough background to follow along through the rest of the book. Query operators were also introduced but will be covered in a lot more detail in Chapter 4 and other sections of the book.

In the next chapter, I provide you with additional tuning tools and techniques such as using SQL Trace, extended events, and DMVs to find out which queries are consuming the most resources or to find out some other performance-related problems.