**Execution Plan Basics**

Every day, out in the various discussion boards devoted to Microsoft SQL Server, the same types of questions come up again and again: Why is this query running slow? Is my index getting used? Why isn't my index getting used? Why does this query run faster than this query?. The correct response is probably different in each case, but in order to arrive at the answer you have to ask the same return question in each case: have you looked at the execution plan?

**What is an execution plan?**

Every day, out in the various discussion boards devoted to Microsoft SQL Server, the same types of questions come up again and again: Why is this query running slow? Is my index getting used? Why isn’t my index getting used? Why does this query run faster than this query?. The correct response is probably different in each case, but in order to arrive at the answer you have to ask the same return question in each case: have you looked at the execution plan? An execution plan, simply put, is the result of the **query optimizer’s**attempt to calculate the most efficient way to impleÂ­ment the request represented by the T-SQL query you subÂ­mitted.

Execution plans can tell you how a query will be executed, or how a query was executed. They are, therefore, the DBA’s primary means of troubleshooting a poorly performing query. Rather than guess at why a given query is performing thousands of scans, putting your I/O through the roof, you can use the execution plan to identify the exact piece of SQL code that is causing the problem. For example, it may be scanning an entire table-worth of data when, with the proper index, it could simply backpack out only the rows you need. All this and more is displayed in the execution plan.

The aim of this chapter is to enable you to capture actual and estimated execution plans, in either graphical, text or XML forÂ­mat, and to understand the basics of how to interpret them. In order to do this, we’ll cover the following topics:

* **A brief backgrounder on the query optimizer –**execution plans are a result of the optimizer’s calculations so it’s useful to know at least a little bit about what the optimizer does, and how it works
* **Actual and Estimated execution plans –**what they are and how they differ
* **Capturing and interpreting the different visual execution plan formats** – we’ll investigate graphical, text and XML execution plans for a very basic SELECT query
* **Automating execution plan capture –**using the SQL Server Profiler tool

What Happens When a Query is Submitted?

When you submit a query to a SQL Server database, a number of processes on the server go to work on that query. The purpose of all these processes is to manage the system such that it will provide your data back to you, or store it, in as timely a manner as possible, whilst maintaining the integrity of the data.

These processes are run for each and every query submitted to the system. While there are lots of different actions occurring simultaneously within SQL Server, we’re going to focus on the processes around T-SQL. They break down roughly into two stages:

1. Processes that occur in the relational engine
2. Processes that occur in the storage engine.

In the relational engine the query is parsed and then processed by the Query Optimizer , which generates an execution plan. The plan is sent (in a binary format) to the storage engine, which it then uses to retrieve or update the underlying data. The storage engine is where processes such as locking, index maintenance and transactions occur. Since **execution plans** are created in the relational engine, that’s where we’ll be focusing our attention.

Query Parsing

When you pass a T-SQL query to the SQL Server system, the first place it goes to is the relÂ­ationÂ­al engine. [[1]](https://www.simple-talk.com/sql/performance/execution-plan-basics/#ftn1)

As the T-SQL arrives, it passes through a process that checks that the T-SQL is written correctly, that it’s well formed. This process is known as query *parsing*. The output of the **Parser** process is a parse tree, or query tree (or even sequence tree). The parse tree represents the logical steps necessary to execute the query that has been requested.

If the T-SQL string is not a data manipulation language (DML) statement, it will be not be optimized because, for example, there is only one “right way” for the SQL Server system to create a table; therefore, there are no opportunities for improving the performance of that type of statement. If the T-SQL string is a DML statement, the parse tree is passed to a process called the **algebrizer**. The algebrizer resolves all the names of the various objects, tables and columns, referred to within the query string. The algebrizer identifies, at the individual column level, all the types (**varchar(50)** versus **nvarchar(25)** and so on) of the objects being accessed. It also determines the location of aggregates (such as **GROUP BY**, and **MAX**) within the query, a process called *aggregate binding*. This algebrizer process is important because the query may have aliases or synonyms, names that don’t exist in the database, that need to be resolved, or the query may refer to objects not in the database.

The algebrizer outputs a binary called the **query processor tree**, which is then passed on to the **query optimizer**.

The Query Optimizer

The query optimizer is essentially a piece of software that “models” the way in which the database relational engine works. Using the query processor tree and the **statistics** it has about the data, and applying the model, it works out what it thinks will be the optimal way to execute the query – that is, it generates an execution plan.

In other words, the optimizer figures out how best to implement the request represented by the T-SQL query you submitted. It decides if the data can be accessed through indexes, what types of joins to use and much more. The decisions made by the optimizer are based on what it calculates to be the cost of a given execution plan, in terms of the required CPU processing and I/O, and how fast it will execute. Hence, this is known as a **cost-based** plan.

The optimizer will generate and evaluate many plans (unless there is already a cached plan) and, generally speaking, will choose the lowest-cost plan i.e. the plan it thinks will execute the query as fast as possible and use the least amount of resources, CPU and I/O. The calculation of the execution speed is the most important calculation and the optimizer will use a process that is more CPU-intensive if it will return results that much faster. Sometimes, the optimizer will select a less efficient plan if it thinks it will take more time to evaluate many plans than to run a less efficient plan.

If you submit a very simple query – for example, a single table with no indexes and with no aggregates or calculations within the query – then rather than spend time trying to calculate the absolute optimal plan, the optimizer will simply apply a single, **trivial plan** to these types of queries.

If the query is non-trivial, the optimizer will perform a cost-based calculation to select a plan. In order to do this, it relies on **statistics** that are maintained by SQL Server.

Statistics are collected on columns and indexes within the database, and describe the data distribution and the uniqueness, or selectivity of the data. The information that makes up statistics is represented by a **histogram**, a tabulation of counts of the occurrence of a particular value, taken from 200 data points evenly distributed across the data. It’s this “data about the data” that provides the information necessary for the optimizer to make its calculations.

If statistics exist for a relevant column or index, then the optimizer will use them in its calculations. Statistics, by default, are created and updated automatically within the system for all indexes or for any column used as a predicate, as part of a **WHERE** clause or **JOIN** **ON** clause. Table variables do not ever have statistics generated on them, so they are always assumed by the optimizer to have a single row, regardless of their actual size. Temporary tables do have statistics generated on them and are stored in the same histogram as permanent tables, for use within the optimizer.

The optimizer takes these statistics, along with the query processor tree , and heuristically determines the best plan. This means that it works through a series of plans, testing different types of join, rearranging the join order, trying different indexes, and so on, until it arrives at what it thinks will be the fastest plan. During these calculations, a number is assigned to each of the steps within the plan, representing the optimizer’s estimation of the amount of time it thinks that step will take. This shows what is called the **estimated** **cost** for that step. The accumulation of costs for each step is the cost for the execution plan itself.

It’s important to note that the estimated cost is just that – an estimate. Given an infinite amount of time and complete, up-to-date statistics, the optimizer would find the perfect plan for executing the query. However, it attempts to calculate the best plan it can in the least amount of time possible, and is obviously limited by the quality of the statistics it has available. Therefore these cost estimations are very useful as measures, but may not precisely reflect reality.

Once the optimizer arrives at an execution plan, the actual plan is created and stored in a memory space known as the **plan** **cache** – unless an identical plan already exists in the cache (more on this shortly, in the section on *Execution* *Plan Reuse*). As the optimizer generates potential plans, it compares them to previously generated plans in the cache. If it finds a match, it will use that plan.

Query Execution

Once the execution plan is generated, the action switches to the storage engine, where the query is actually executed, according to the plan.

We will not go into detail here, except to note that the carefully generated execution may be subject to *change* during the actual execution process. For example, this might happen if:

* A determination is made that the plan exceeds the threshold for a parallel execution (an execution that takes advantage of multiple processors on the machine – more on parallel execution in the book).
* The statistics used to generate the plan were out of date, or have changed since the original execution plan was created by the optimizer.

The results of the query are returned to you after the relational engine changes the format to match that requested in your T-SQL statement, assuming it was a **SELECT**.

Estimated and Actual Execution Plans

As discussed previously, there are two distinct types of execution plan. First, there is the plan that represents the output from the optimizer. This is known as an **Estimated execution plan**. The operators, or steps, within the plan will be labeled as logical, because they’re representative of the optimizer’s view of the plan.

Next is the plan that represents the output from the actual query execution. This type of plan is known, funnily enough, as the **Actual execution plan**. It shows what actually happened when the query executed.

Execution Plan Reuse

It is expensive for the Server to generate execution plans so SQL Server will keep and reuse plans wherever possible. As they are created, plans are stored in a section of memory called the **plan cache**).

When a query is submitted to the server, an *estimated* execution plan is created by the optimizer. Once that plan is created, and before it gets passed to the storage engine, the optimizer compares this estimated plan to *actual* execution plans that already exist in the plan cache . If an actual plan is found that matches the estimated one, then the optimizer will reuse the existing plan, since it’s already been used before by the query engine. This reuse avoids the overhead of creating actual execution plans for large and complex queries or even simple plans for small queries called thousands of times in a minute.

Each plan is stored once, unless the cost of the plan lets the optimizer know that a parallel execution might result in better performance (more on parallelism in Chapter 8). If the optimizer sees parallelism as an option, then a second plan is created and stored with a different set of operations to support parallelism. In this instance, one query gets two plans.

Execution plans are not kept in memory forever. They are slowly aged out of the system using an “age” formula that multiplies the estimated cost of the plan by the number of times it has been used (e.g. a plan with a cost of 10 that has been referenced 5 times has an “age” value f of 50). The lazywriter process, an internal process that works to free all types of cache (including plan cache ), periodically scans the objects in the cache and decreases this value by one each time.

If the following criteria are met, the plan is removed from memory:

* More memory is required by the system
* The “age” of the plan has reached zero
* The plan isn’t currently being referenced by an existing connection

Execution plans are not sacrosanct. Certain events and actions can cause a plan to be recompiled. It is important to remember this because recompiling execution plans can be a very expensive operation. The following actions can lead to recompilation of an execution plan:

* Changing the structure or schema of a table referenced by the query
* Changing an index used by the query
* Dropping an index used by the query
* Updating the statistics used by the query
* Calling the function, **sp\_recompile**
* Subjecting the keys in tables referenced by the query to a large number of inserts or deletes
* For tables with triggers, significant growth of the **inserted** or **deleted** tables
* Mixing DDL and DML within a single query, often called a deferred compile
* Changing the **SET** options within the execution of the query
* Changing the structure or schema of temporary tables used by the query
* Changes to dynamic views used by the query
* Changes to cursor options within the query
* Changes to a remote rowset, like in a distributed partitioned view
* When using client side cursors, if the **FOR BROWSE** options are changed

Since the cache plays such an important role in how execution plans operate, you need a few tools for querying and working with the plan cache . First off, while testing, you may want to see how long a plan takes to compile, or to investigate how minor adjustments might create slightly different plans. To completely clear the cache, run this:

|  |  |
| --- | --- |
|  | DBCC FREEPROCCACHE |

You’re going to want to see the objects within the cache in order to see how the optimizer and storage engine created your plan. With dynamic management views and dynamic management functions, we can easily put together a query to get a very complete set of information about the execution plans on our system:

|  |  |
| --- | --- |
|  | SELECT [cp].[refcounts]  , [cp].[usecounts]  , [cp].[objtype]  , [st].[dbid]  , [st].[objectid]  , [st].[text]  , [qp].[query\_plan]  FROM sys.dm\_exec\_cached\_plans cp  CROSS APPLY sys.dm\_exec\_sql\_text ( cp.plan\_handle ) st  CROSS APPLY sys.dm\_exec\_query\_plan ( cp.plan\_handle ) qp ; |

With this query we can see the SQL called and the XML plan generated by the execution of that SQL. You can use the XML directly or open it as a graphical execution plan.

Why the Actual and Estimated Execution Plans Might Differ

Generally, you probably won’t see any differences between your estimated and actual execution plans. However, circumÂ­stances can arise that can cause differences between the estiÂ­mated and actual execution plans.

When Statistics are Stale

The main cause of a difference between the plans is differÂ­ences between the *statistics* and the actual data. This genÂ­erÂ­ally occurs over time as data is added and deleted. This causes the key values that define the index to change, or their distribution (how many of what type) to change. The automatic update of statistics that occurs, assuming it’s turned on, only samples a subset of the data in order to reduce the cost of the operation. This means that, over time, the statistics become a less-and-less accurate reflection of the actual data. Not only can this cause diffÂ­erÂ­ences between the plans, but you can get bad execution plans because the statistical data is not up to date. [[2]](https://www.simple-talk.com/sql/performance/execution-plan-basics/#ftn2)

When the Estimated Plan is Invalid

In some instances, the estimated plan won’t work at all. For example, try generating an estimated plan for this simple bit of code:

|  |  |
| --- | --- |
|  | CREATE TABLE TempTable  (  Id INT IDENTITY (1 , 1 )  ,Dsc NVARCHAR (50 )  );    INSERT INTO TempTable ( Dsc )  SELECT [Name]  FROM [Sales] .[Store] ;    SELECT \*  FROM TempTable ;    DROP TABLE TempTable ; |

You will get this error:

|  |  |
| --- | --- |
|  | Msg 208, Level 16, State 1, Line 7    Invalid object name 'TempTable'. |

The optimizer, which is what is used to generate Estimated Execution plans, doesn’t execute T-SQL. It does run the stateÂ­ments through the algebrizer , the process outlined earlier that is responsible for verifying the names of database objects. Since the query has not yet been executed, the temporary table does not yet exist. This is the cause of the error. Running this same bit of code through the Actual execution plan will work perfectly fine.

When Parallelism is Requested

When a plan meets the threshold for parallelism (more about this in Chapter 8) two plans are created. Which plan is actually executed is up to the query engine. So you might see a plan with, or without, parallel operators in the estimated execution plan. When the query actually executes, you may see a completely different plan if the query engine determines that it either can’t support a parallel query at that time or that a parallel query is called for.

Execution Plan Formats

SQL Server offers only one type of execution plan (be it estimated or actual), but three different formats in which to view that execution plan.

* Graphical Plans
* Text Plans
* XML Plans

The one you choose will depend on the level of detail you want to see, and on the individual DBA’s preferences and methÂ­ods.

Graphical Plans

These are quick and easy to read but the detailed data for the plan is masked. Both Estimated and Actual execution plans can be viewed in graphical format.

Text Plans

These are a bit harder to read, but more information is immediately available. There are three text plan formats:

* **SHOWPLAN\_ALL**: a reasonably complete set of data showing the Estimated execution plan for the query
* **SHOWPLAN\_TEXT** : provides a very limited set of data for use with tools like **osql.exe**. It too only shows the Estimated execution plan
* **STATISTICS PROFILE:**similar to **SHOWPLAN\_ALL**except it represents the data for the Actual execution plan

XML Plans

XML plans present the most complete set of data available on a plan, all on display in the structured XML format. There are two varieties of XML plan:

* **SHOWPLAN\_XML**: The plan generated by the optimizer prior to execution.
* **STATISTICS\_XML** : The XML format of the Actual execution plan.

Getting Started

Execution plans are there to assist you in writing efficient T-SQL code, troubleshooting existing T-SQL behavior or monitoring and reporting on your systems. How you use them and view them is up to you, but first you need to understand the information contained within the plans and how to interpret it. One of the best ways to learn about execution plans is to see them in action, so let’s get started.

Please note that occasionally, especially when we move on to more complex plans, the plan that you see may differ slightly from the one presented in the book. This might be because we are using different versions of SQL Server (different SP levels and hot fixes), that we are using slightly different versions of the AdventureWorks database, or because of how the AdventureWorks database has been altered over time as each of us has played around in it. So while most of the plans you get should be very similar to what we display here, don’t be too surprised if you try the code and see something different.

Sample Code

Throughout the following text, I’ll be supplying T-SQL code that you’re encouraged to run for yourself. All of the source code is freely downloadable from the **Simple Talk Publishing**website (http://www.simpletalkpublishing.com).

The examples are written for SQL 2005 sample database, **Adventureworks**. You can get hold of get a copy of **Adventureworks** from here:

<http://www.codeplex.com/MSFTDBProdSamples>

If you are working with procedures and scripts other than those supplied, please remember that encrypted procedures will not display an execution plan.

The plans you see may not precisely reflect the plans generated for the book. Depending on how old a given copy of AdventureWorks may be, the statistics could be different, the indexes may be different, the structure and data may be different. So please be aware that you won’t always see the same thing if you run the examples.

The initial execution plans will be simple and easy to read from the samples presented in the text. As the queries and plans become more complicated, the book will describe the situation but, in order to easily see the graphical execution plans or the complete set of XML, it will be necessary to generate the plans. So, please, read next to your machine, so that you can try running each query yourself!

Permissions Required to View Execution Plans

In order to see the execution plans for the following queries you must have the correct permissions within the database. Once that’s set, assuming you’re not **sysadmin**, **dbcreator**or **db\_owner**, you’ll need to be granted the **ShowPlan** permission within the database being tested. Further, you’ll need this permission on each database referenced by the queries for which you hope to generate a plan. Run the statement:

|  |  |
| --- | --- |
|  | GRANT SHOWPLAN TO [username] |

Substituting the user name will enable execution plans for that user on that database.

Working with Graphical Execution Plans

In order to focus on the basics of capturing Estimated and Actual execution plans, the first query will be one of the simplest possible queries, and we’ll build from there. Open up Management Studio, and type the following into the query window:

|  |  |
| --- | --- |
|  | SELECT \*  FROM [dbo].[DatabaseLog]; |

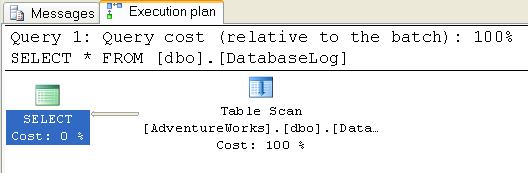
Getting the Estimated Plan

We’ll start by viewing the graphical **estimated execution plan** that is generated by the query optimizer, so there’s no need to actually run the query yet.

We can find out what the optimizer estimates to be the least costly plan in one of following ways:

* Click on the “Display Estimated Execution Plan icon on the tool bar.
* Right-click the query window and select the same option from the menu.
* Click on the Query option in the menu bar and select the same choice.
* Simply hit CTRL-L on the keyboard.

I tend to click the icon more often than not but, either way, we see our very first **Estimated execution plan**, as in Figure 1.

****

*Figure 1*

We’ll explain what this plan means shortly, but first, let’s capture the Actual execution plan.

Getting the Actual Plan

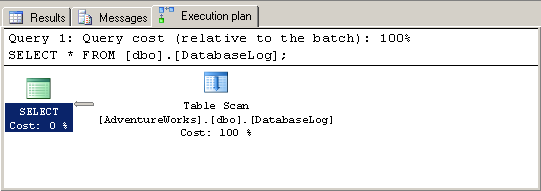
Actual execution plans, unlike Estimated execution plans, do not represent the calculations of the optimizer. Instead this execution plan shows what happened when the query was executed. The two will often be identical but will sometimes differ, due to changes to the execution plan made by the storage engine.

Again, there are several ways to generate our first graphical Actual Execution Plan :

* Click on the icon on the tool bar called “Include Actual Execution Plan
* Right-click within the query window and choose the “Include Actual Execution Plan menu item.
* Choose the same option in the Query menu choice.
* Type Control-M.

Each of these methods functions as an “on” switch and an execution plan will be created for all queries run from that query window until you turn it off again.

So, activate execution plans by your preferred method and execute the query. You should see an execution plan like the one in Figure 2.

****

*Figure 2*

In this simple case the Actual plan is identical to the EstimaÂ­ted plan.

Interpreting Graphical Execution Plan

The icons you see in Figures 1 and 2 are the first two of appÂ­roxÂ­imately 78 operators that represent various actions and decisÂ­ions that potentially make up an execution plan. On the left is the SELECT icon, an icon that you’ll see quite a lot of and that you can usually completely ignore. It’s the final result and formatting from the relational engine. The icon on the right represents a **table scan**[**[3]**](https://www.simple-talk.com/sql/performance/execution-plan-basics/#ftn3). This is the first, and one of the easiest, icons to look for when trying to track down performance problems.

Usually, you read a graphical execution plan from right to left and top to bottom. You’ll also note that there is an arrow pointing between the two icons. This arrow represents the data being passed between the operators, as represented by the icons. So, in this case, we simply have a table scan operator producing the result set (represented by the Select operator). The thickness of the arrow reflects the amount of data being passed, thicker meaning more rows. This is another visual clue as to where performance issues may lie. You can hover with the mouse pointer over these arrows and it will show the number of rows that it represents. For example, if your query returns two rows, but the execution plan shows a big thick arrow indicating many rows being processed, then that’s something to possibly investigate.

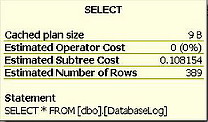
Below each icon is displayed a number as a percentage. This number represents the relative cost to the query for that operator. That cost, returned from the optimizer, is the estimated execution time for that operation. In our case, all the cost is associated with the table scan. While a cost may be represented as 0% or 100%, remember that, as these are ratios, not actual numbers, even a 0% operator will have a small cost associated with it.

Above the icons is displayed as much of the query string as will fit and a “cost (relative to batch)” of 100%. Just as each query can have multiple steps, and each of those steps will have a cost relative to the query, you can also run multiple queries within a batch and get execution plans for them. They will then show up as different costs as a part of the whole.

ToolTips

Each of the icons and the arrows has, associated with it, a pop-up window called a **ToolTip**, which you can access by hovering your mouse pointer over the icon.

Pull up the Estimated execution plan, hover over the **SELECT** operator, and you should see the ToolTip window shown in Figure 3.

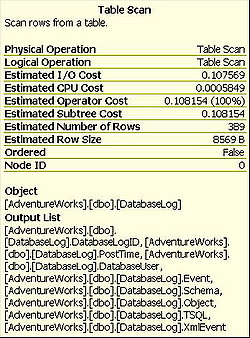
****

*Figure 3*

Here we get the numbers generated by the optimizer on the following:

* **Cached plan size** – how much memory the plan generated by this query will take up in stored procedure cache. This is a useful number when investigating cache performance issues because you’ll be able to see which plans are taking up more memory.
* **Estimated Operator Cost –**we’ve already seen this as the percentage cost in Figure 1.
* **Estimated Subtree Cost** – tells us the accumulated optimizer cost assigned to this step and all previous steps, but remember to read from right to left. This number is meaningless in the real world, but is a mathematical evaluation used by the query optimizer to determine the cost of the operator in question; it represents the amount of time that the optimizer thinks this operator will take.
* **Estimated number of rows** – calculated based on the statistics available to the optimizer for the table or index in question.

Below this information, we see the statement that represents the entire query that we’re processing. If we look at the ToolTip information for the Table Scan we see the information in Figure 4.

****

*Figure 4*

Each of the different operators will have a distinct set of data. The operator in Figure 4 is performing work of a different nature than that in Figure 3, and so we get a different set of details. First, the Physical and Logical Operations are listed. The logical operators are the results of the optimizer’s calculations for what should happen when the query executes. The physical operators represent what actually occurred. The logical and physical operators are usually the same, but not always – more on that in the book.

After that, we see the estimated costs for I/O, CPU, Operator and Subtree. The Subtree is simply the section of the execution tree that we have looked at so far, working right to left again, and top to bottom. All estimations are based on the statistics available on the columns and indexes in any table.

The I/O Cost and CPU cost are not actual operators, but rather the cost numbers assigned by the Query Optimizer during its calculations. These numbers are useful when determining whether most of the cost is I/O-based (as in this case), or if we’re putting a load on the CPU. A bigger number means more processing in this area. Again, these are not hard and absolute numbers, but rather pointers that help to suggest where the actual cost in a given operation may lie.

You’ll note that, in this case, the operator cost and the subtree cost are the same, since the table scan is the only operator. For more complex trees, with more operators, you’ll see that the cost accumulates as the individual cost for each operator is added to the total. You get the full cost of the plan from the final operation in the query plan, in this case the **Select** operator.

Again we see the estimated number of rows. This is displayed for each operation because each operation is dealing with different sets of data. When we get to more complicated execution plans, you’ll see the number of rows change as various operators perform their work on the data as it passes between each operator. Knowing how the rows are added or filtered out by each operator helps you understand how the query is being performed within the execution process.

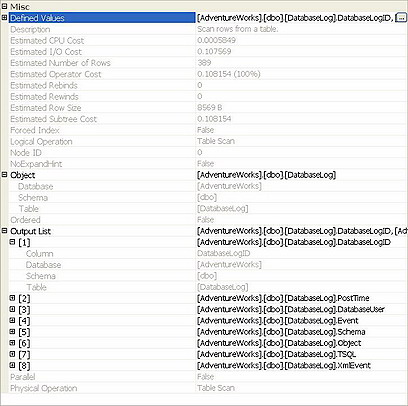
Another important piece of information, when attempting to troubleshoot performance issues, is the Boolean value displayed for **Ordered**. This tells you whether or not the data that this operator is working with is in an ordered state. Certain operations, for example, an **ORDER BY** clause in a **SELECT** statement, may require data to be placed in a particular order, sorted by a particular value or set of values. Knowing whether or not the data is in an **Ordered** state helps show where extra processing may be occurring to get the data into that state.

Finally, **Node ID** is the ordinal, which simply means numbered in order, of the node itself, interestingly enough numbered left to right, despite the fact that the operations are best read right to left.

All these details are available to help you understand what’s happening within the query in question. You’ll be able to walk through the various operators, observing how the subtree cost accÂ­umuÂ­lates, how the number of rows changes, and so on. With these details you’ll be able to identify processes that are using excessive amounts of CPU or tables that need more indexes, or indexes that are not used, and so on.

Operator Properties

More information is available than that presented in the ToolTips. Right-click any icon within a graphical execution plan and select the “Properties” menu item to get a detailed list of information about that operation. Figure 5 shows the details from the original table scan.

****

*Figure 5*

Most of this information should be familiar, but some of it is new. Starting from the top, **Defined Values**displays the inforÂ­mation that this operation adds to the process. These can be a part of the basic query, in this case the columns being selected, or they can be internally created values as part of the query processing, such as a flag used to determine referential integrity, or a placeholder for counts for aggregate functions.

Under the **Defined Values**we get a description of the operation and then some familiar **Estimated Cost**data. After that we see:

* **Estimated Rebinds**and**Rewinds**, values which describe the number of times an **Init()** operator is called in the plan.
* The**Forced Index** value would be **True** when a query hint is used to put a specific index to use within a query. SQL Server supplies the functionality in query hints as a way to give you some control over how a query is executed. Query hints are covered in detail in the book
* **NoExpandHint** this is roughly the same concept as Forced Index, but applied to indexed views.

By expanding the **Object** property, you can see details on the object in question. The **Output List** property provides details of each of the output columns. You’ll also find out whether or not this operator is taking part in a parallel operation, (when multiple CPUs are used by one operator).

Working with Text Execution Plan

The graphical execution plans are very useful because they’re so easy to read. However, a lot of the data about the operators is not immediately visible to you. Some can be seen in a limited form in the ToolTip windows, and the complete set is available in the Properties window. Wouldn’t it be great if there was a way to see all that information at once?

In the case of really large queries with incredibly complex plans or large number of batch statements, wouldn’t it be handy to be able to search through for particular bits of information, table scans or the highest operator cost or something? Well, you can. Two methods exist: Text Execution Plan .

Microsoft is planning on deprecating Text Execution Plan , so we’ll cover them in relatively little detail.

Getting the Estimated Text Plan

To activate the text version of the Estimated text execution plan, simply issue the following command at the start of the query:

|  |  |
| --- | --- |
|  | SET SHOWPLAN\_ALL ON; |

It’s important to remember that, with SHOWPLAN\_ALL set to ON, execution information is collected for all subsequent T-SQL statements, but those statements are not actually executed. Hence, we get the estimated plan. It’s very important to remember to turn **SHOWPLAN\_ALL OFF** after you have captured the information you require. If you forget, and submit a **CREATE**, **UPDATE** or **DELETE** statement with **SHOWPLAN\_ALL** turned on, then those statements *won’t* be executed, and a table you might expect to exist, for example, will not.

To turn **SHOWPLAN\_ALL**off, simply issue:

|  |  |
| --- | --- |
|  | SET SHOWPLAN\_ALL ; |

We can also use the equivalent commands for **SHOWPLAN\_TEXT**. The text-only show plan is meant for use with tools like **osql.exe**, where the result sets can be readily parsed and stored by a tool dealing with text values, as opposed to actual result sets, as the **SHOWPLAN\_ALL** function does.

We focus only on **SHOWPLAN\_ALL**here.

Getting the Actual Text Plan

In order to activate and deactivate the text version of the Actual execution plan, use:

|  |  |
| --- | --- |
|  | SET STATISTICS PROFILE ON |

And:

|  |  |
| --- | --- |
|  | SET STATISTICS PROFILE OFF |

Interpreting Text Plan

We’ll stick with the same basic query we used when discussing graphical plans, so execute the following:

|  |  |
| --- | --- |
|  | GO  SELECT \*  FROM [dbo].[DatabaseLog] ;  GO  SET SHOWPLAN\_ALL OFF ;  GO |

When you execute this query, the estimated plan is shown in the results pane. Here is the first column of the results:

**504-image011.png**

Figure 6

This screen shot was trimmed to keep the text as readable as possible. The text plan generated roughly parallels the graphical plan. The first row is the **SELECT** statement that was submitted. The rows following are the physical operations occurring within the query plan. In or case that means one row i.e. the table scan.

As we progress and view more complex text plans, in the book, you’ll quickly realize that they are not as readily readable as the graphical plan. There’s also no easy route through the query, such as we have with the “read it right to left” approach in the graphical plans. You start in the middle and move outwards, helped by the indentation of the data and the use of pipe ( | ) to connect the statements parent to child.

In addition to the first column, the details that were hidden in the ToolTip or in the Properties window are displayed in a series of columns. Most of the information that you’re used to seeing is here, plus a little more. So, while the **NodeId** was available in the graphical plan, because of the nature of the graphical plan, nothing was required to identify the parent of a given node. In the **SHOWPLAN\_ALL** we get a column showing the **Parent** NodeId. As you scan right you’ll see many other familiar columns, such as the **TotalSubTreeCost**, **EstimateRows** and so on. Some of the columns are hard to read, such as the Defined List (the values or columns introduced by this operation to the data stream), which is displayed as just a comma-separated list .

Working with XML Execution Plans

XML Plans are the new and recommended way of displaying the execution plans in SQL Server 2005. They offer functionality not previously available.

Getting the Actual and Estimated XML Plan

In order to activate and deactivate the XML version of the Estimated execution plan, use:

|  |  |
| --- | --- |
|  | SET SHOWPLAN\_XML ON  ...  SET SHOWPLAN\_XML OFF |

As for **SHOWPLAN\_ALL**command is essentially an instruction not to execute any T-SQL statements that follow, but instead to collect execution plan information for those statements, in the form of an XML document. Again, it’s important to turn SHOWPLAN\_XML off as soon as you have finished collecting plan information, so that subsequent T-SQL execute as intended.

For the XML version of the Actual plan, use:

|  |  |
| --- | --- |
|  | SET STATISTICS XML ON  ...  SET STATISTICS XML OFF |

Interpreting XML Plan

Once again, let’s look at the same execution plan as we evaluated with the text plan.

|  |  |
| --- | --- |
|  | GO  SET SHOWPLAN\_XML ON ;  GO  SELECT \*  FROM [dbo] .[DatabaseLog] ;  SET SHOWPLAN\_XML OFF ;  GO |

The result, in the default grid mode, is shown in figure 7:



*Figure 7*

The link is a pointer to an XML file located here:

|  |  |
| --- | --- |
|  | \Microsoft SQL Server\90\Tools\Binn\schemas\sqlserver\2003\03\showplan\showplanxml.xsd |

Clicking on this link opens the execution plan in XML format in a browser window within the SQL Server Management Studio (SSMS). You can view the output in text, grid or file (default is grid). You can change the output format from the **Query** | **Results To**menu option.

A lot of information is put at your fingertips with XML plans – much of which we won’t encounter here with our simple example, but will get to in later, more complex plans. Nevertheless, even this simple plan will give you a good feel for the XML format.

The results, even for our simple query, are too large to output here. I’ll go over them by reviewing various elements and attriÂ­butes. The full schema is available here:

|  |  |
| --- | --- |
|  | http://schemas.microsoft.com/sqlserver/2004/07/showplan/. |

Listed first are the **BatchSequence** , **Batch** and **Statements** elements. In this example, we’re only looking at a single Batch and a single Statement, so nothing else is displayed. Next, like all the other execution plans we’ve reviewed so far, we see the query in question as part of the **StmtSimple** element. Within that, we receive a list of attributes of the statement itself, and some physical attributes of the **QueryPlan**:

|  |  |
| --- | --- |
|  | < StmtSimpleStatementText="SELECT \*  FROM [dbo].[DatabaseLog];  " StatementId="1" StatementCompId="1" StatementType="SELECT" StatementSubTreeCost="0.108154" StatementEstRows="389" StatementOptmLevel="TRIVIAL">  < StatementSetOptions QUOTED\_IDENTIFIER="false" ARITHABORT="true" CONCAT\_NULL\_YIELDS\_NULL="false" ANSI\_NULLS="false" ANSI\_PADDING="false" ANSI\_WARNINGS="false" NUMERIC\_ROUNDABORT="false" />  < QueryPlan CachedPlanSize="9"> |

Clearly a lot more information is on immediate display than was provided for **SHOWPLAN\_ALL**. Notice that the optimizer has chosen a trivial execution plan, as we might expect. Information such as the **CachedPlanSize** will help you to determine if, for example, your query exceeds one page in length, and gets sent into the **LeaveBehind** memory space.

After that, we have the **RelOp** element, which provides the information we’re familiar with, regarding a particular operation, in this case the table scan.

|  |  |
| --- | --- |
|  | < RelOp NodeId="0" PhysicalOp="Table Scan" LogicalOp="Table Scan" EstimateRows="389" EstimateIO="0.107569" EstimateCPU="0.0005849" AvgRowSize="8569" EstimatedTotalSubtreeCost="0.108154" Parallel="0" EstimateRebinds="0" EstimateRewinds="0"> |

Not only is there more information than in the text plans, but it’s also much more readily available and easier to read than in either the text plans or the graphical plans (although the flow through the graphical plans is much easier to read). For example, a problematic column, like the Defined List mentioned earlier, that is difficult to read, becomes the **OutputList** element with a list of **ColumnReference** elements, each containing a set of attributes to describe that column:

|  |  |
| --- | --- |
|  | <OutputList>  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="DatabaseLogID" />  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="PostTime" />  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="DatabaseUser" />  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="Event" />  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="Schema" />  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="Object" />  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="TSQL" />  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="XmlEvent" />  </ OutputList> |

This makes XML not only easier to read, but much more readily translated directly back to the original query.

Back to the plan, after **RelOp**element referenced above we have the table scan element:

|  |  |
| --- | --- |
|  | < TableScan Ordered="0" ForcedIndex="0" NoExpandHint="0"> |

Followed by a list of defined values that lays out the columns referenced by the operation:

|  |  |
| --- | --- |
|  | < DefinedValues>  < DefinedValue>  < ColumnReference Database="[AdventureWorks]" Schema="[dbo]" Table="[DatabaseLog]" Column="DatabaseLogID" />  </ DefinedValue>  < DefinedValue>  ...<output cropped>........ |

Saving XML Plans as Graphical Plan

You can save the execution plan without opening it by right-clicking within the results and selecting “Save As.” You then have to change the filter to “\*.\*” and when you type the name of the file you want to save add the extension “.sqlplan.” This is how the Books Online recommends saving an XML execution plan. In fact, what you get when you save it this way is actually a **graphical execution plan** file. This can actually be a very useful feature. For example, you might collect multiple plans in XML format, save them to file and then open them in easy-to-view (and compare) graphical format.

One of the benefits of extracting an XML plan and saving it as a separate file is that you can share it with others. For example, you can send the XML plan of a slow-running query to a DBA friend and ask them their opinion on how to rewrite the query. Once the friend receives the XML plan, they can open it up in Management Studio and review it as a graphical execution plan.

In order to actually save an XML plan as XML, you need to first open the results into the XML window. If you attempt to save to XML directly from the result window you only get what is on display in the result window. Another option is to go to the place where the plan is stored, as defined above, and copy it.

Automating Plan Capture Using SQL Server Profiler

During development you will capture execution plans for targeted T-SQL statements, using one of the techniques described in this chapter. You will activate execution plan capture, run the query in question, and then disable it again.

However, if you are troubleshooting on a test or live production server, the situation is different. A production system may be subject to tens or hundreds of sessions executing tens or hundreds or queries, each with varying parameter sets and varying plans. In this situation we need a way to automate plan capture so that we can collect a large number of plans simultaneously. In SQL Server 2005 you can use Profiler to capture XML execution plans, as the queries are executing. You can then examine the collected plans, looking for the queries with the highest costs, or simply searching the plans to find, for example, Table Scan operations that you’d like to eliminate.

SQL Server 2005 Profiler is a powerful tool that allows you to capture data about events, such as the execution of T-SQL or a stored procedure, occurring within SQL Server. Profiler events can be tracked manually, through a GUI interface, or traces can be defined through T-SQL (or the GUI) and automated to run at certain times and for certain periods.

These traces can be viewed on the screen or sent to or to a file or a table in a database.[[4]](https://www.simple-talk.com/sql/performance/execution-plan-basics/#ftn4)

Execution Plan events

The various trace events that will generate an execution plan are as follow:

* **Showplan Text** : This event fires with each execution of a query and will generate the same type of estimated plan as the **SHOWPLAN\_TEXT** T-SQL statement. Showplan Text will work on SQL 2005 databases, but it only shows a subset of the information available to ShowPlan XML. We’ve already discussed the shortcomings of the text execution plans, and this is on the list for deprecation in the future.
* **Showplan Text (unencoded)** : Same as above, but it shows the information as a string instead of binary. This is also on the list for deprecation in the future.
* **Showplan All** : This event fires as each query executes and will generate the same type of estimated execution plan as the **SHOWPLAN\_ALL** TSQL statement. This has the same shortcomings as Showplan Text, and is on the list for future deprecation.
* **Showplan All for Query Compile** : This event generates the same data as the Showplan All event, but it only fires when a query compile event occurs. This is also on the list for deprecation in the future.
* **Showplan Statistics Profile** : This event generates the actual execution plan in the same way as the TSQL command **STATISTICS PROFILE**. It still has all the shortcomings of the text output, including only supplying a subset of the data available to **STATISTICS XML**in TSQL or the **Showplan XML Statistics Profile**event in SQL Server Profiler . The **Showplan Statistics Profile** event is on the list for deprecation.
* **Showplan XML** : The event fires with each execution of a query and generates an estimated execution plan in the same way as **SHOWPLAN\_XML**.
* **Showplan XML For Query Compile** : Like Showplan XML above, but it only fires on a compile of a given query.
* **Performance Statistics** : Similar to the Showplan XML For Query Compile event, except this event captures performance metrics for the query as well as the plan. This only captures XML output for certain event subclasses, defined with the event. It fires the first time a plan is cached, compiled, recompiled or removed from cache.
* **Showplan XML Statistics Profile** : This event will generate the actual execution plan for each query, as it runs.

Capturing all of the execution plans, using Showplan XML or Showplan XML Statistics Profile, inherently places a sizeable load on the server. These are not lightweight event capture scenarios. Even the use of the less frequent Showplan XML for Query Compile will cause a small performance hit. Use due diligence when running traces of this type against any production machine.

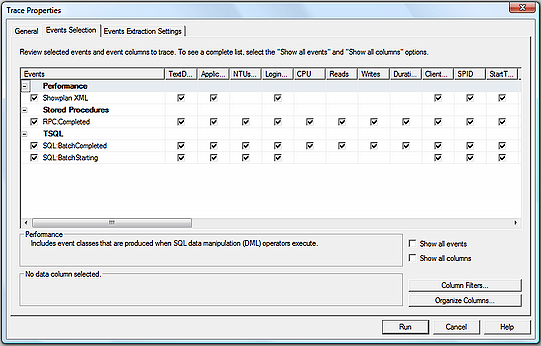
Capturing a Showplan XML Trace

The SQL Server 2005 Profiler Showplan XML event captures the XML execution plan used by the query optimizer to execute a query. To capture a basic Profiler trace, showing estimated execution plans, start up Profiler, create a new trace and connect to a server [[5]](https://www.simple-talk.com/sql/performance/execution-plan-basics/#ftn5).

Switch to the “Events Selection” tab and click on the “Show all events” check box. The Showplan XML event is located within the Performance section, so click on the plus (+) sign to expand that selection. Click on the Showplan XML event.

While you can capture the Showplan XML event by itself in Profiler, it is generally more useful if you capture it along with some other basic events, such as:

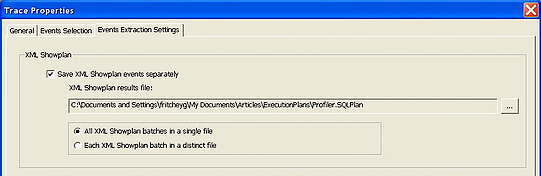
* RPC: Completed
* SQL:BatchStarting
* SQL:BatchCompleted

****

*Figure 8*

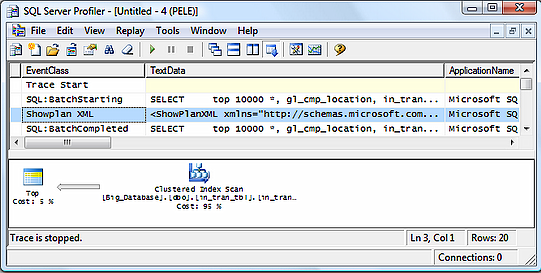
These extra events provide additional information to help put the XML plan into context. For example, you can see what occurred just before and after the event you are interested in.

Once Showplan XML is selected, or any of the other XML events, a third tab appears called **Events Extraction Settings**. On this tab, you can choose to output the XML as it’s generated to a separate file, for later use. Not only can you define the file, but also determine whether or not all the XML will go into a single file or a series of files, unique to each execution plan.

****

*Figure 9*

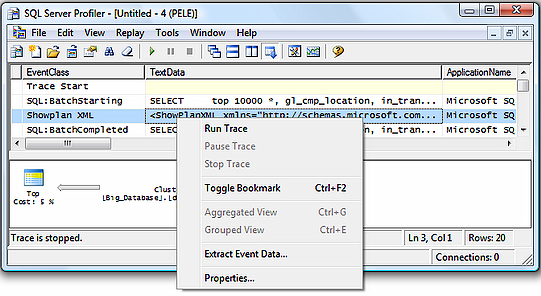
Click on the “Run” button in order to start the trace. When you capture the above events, you get a trace like the one shown in Figure 10.

****

*Figure 10*

Notice that I have clicked on the Showplan XML event. Under the **TextData**column, you see the actual XML plan code. While you can’t see all of it in the screen shot above, it is all there and can be saved to an individual file. In the second window, you can see the graphical execution plan, which is how most people prefer to read and analyze execution plans. So, in effect, the Showplan XML event available in Profiler not only shows you the XML plan code, but also the graphical execution plan.

At this stage, you can also save the code for this particular Showplan XML event to a separate file. Simply right-click on the Showplan XML event you want to save, then select “Extract Event Data.”

****

*Figure 11*

This brings up a dialog box where you can enter the path and filename of the XML code you want to store. Instead of storing the XML code with the typical XML extension, the extension used is .SQLPlan. By using this extension, when you double-click on the file from within Windows Explorer, the XML code will open up in Management Studio in the form of a graphical execution plan.

Whether capturing Estimated execution plans or Actual execution plans, the Trace events operate in the same manner as when you run the T-SQL statements through the query window within Management Studio. The main difference is that this is automated across a large number of queries, from ad-hoc to stored procedures, running against the server.