Collecting and Storing Poor Performing SQL Server Queries for Analysis

By: Ben Snaidero | Read Comments (11) | **Related Tips:** More > Performance Tuning

Problem

In an ideal world all of our queries would be optimized before they ever make it to a production SQL Server environment, but this is not always the case. Smaller data sets, different hardware, schema differences, etc. all effect the way our queries perform. This tip will look at a method of automatically collecting and storing poor performing SQL statements so they can be analyzed at a later date.

Solution

With the new Dynamic Management Views and functions available starting in SQL Server 2005, capturing information regarding the performance of you SQL queries is a pretty straightforward task. The following view and functions give you all the information you need to determine how the SQL in you cache is performing:

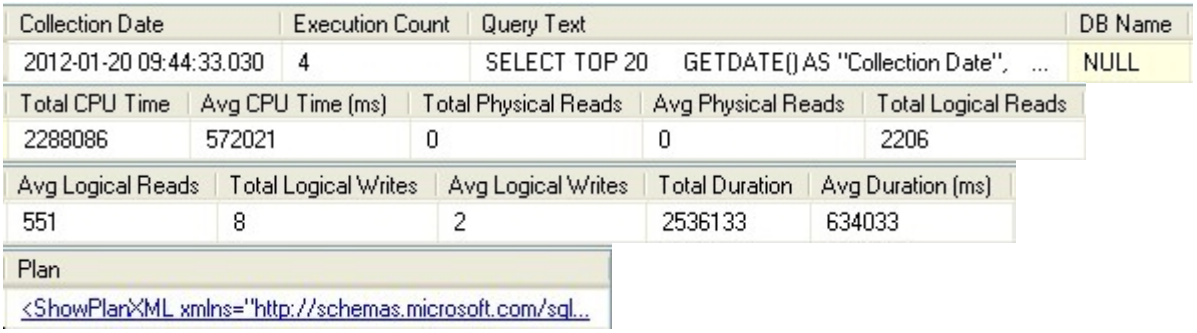
- sys.dm_exec_query_stats
- sys.dm_exec_sql_text(sql_handle)
- sys.dm_exec_query_plan(plan_handle)

Using the view and functions above we can create a query that will pull out all the SQL queries that are currently in the cache. Along with the query text and plan we can also extract some important statistics on the performance of the query as well as the resources used during execution. Here is the query:

```
SELECT TOP 20
    GETDATE() AS "Collection Date",
    qs.execution_count AS "Execution Count",
    SUBSTRING(qt.text,qs.statement_start_offset/2 +1,
                 (CASE WHEN qs.statement_end_offset = -1
                       THEN LEN(CONVERT(NVARCHAR(MAX), qt.text)) * 2
                       ELSE qs.statement_end_offset END -
                            qs.statement_start_offset
                 )/2
             ) AS "Query Text",
     DB_NAME(qt.dbid) AS "DB Name",
     qs.total_worker_time AS "Total CPU Time",
     qs.total_worker_time/qs.execution_count AS "Avg CPU Time (ms)",
     qs.total_physical_reads AS "Total Physical Reads",
```

```
     qs.total_physical_reads/qs.execution_count AS "Avg Physical Reads",
     qs.total_logical_reads AS "Total Logical Reads",
     qs.total_logical_reads/qs.execution_count AS "Avg Logical Reads",
     qs.total_logical_writes AS "Total Logical Writes",
     qs.total_logical_writes/qs.execution_count AS "Avg Logical Writes",
     qs.total_elapsed_time AS "Total Duration",
     qs.total_elapsed_time/qs.execution_count AS "Avg Duration (ms)",
     qp.query_plan AS "Plan"
FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS qt
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) AS qp
WHERE
     qs.execution_count > 50 OR
     qs.total_worker_time/qs.execution_count > 100 OR
     qs.total_physical_reads/qs.execution_count > 1000 OR
     qs.total_logical_reads/qs.execution_count > 1000 OR
     qs.total_logical_writes/qs.execution_count > 1000 OR
     qs.total_elapsed_time/qs.execution_count > 1000
ORDER BY
     qs.execution_count DESC,
     qs.total_elapsed_time/qs.execution_count DESC,
     qs.total_worker_time/qs.execution_count DESC,
     qs.total_physical_reads/qs.execution_count DESC,
     qs.total_logical_reads/qs.execution_count DESC,
     qs.total_logical_writes/qs.execution_count DESC
```

This query can be easily modified to capture something specific if you are looking to solve a particular problem. For example, if you are currently experiencing an issue with CPU on you SQL instance you could alter the WHERE clause and only capture SQL queries where the worker_time is high. Similarly, if you were having an issue with IO, you could only capture SQL queries where the reads or writes are high. Note: The ORDER BY clause is only needed if you keep the TOP parameter in your query. For reference I've included below an example of the output of this query.



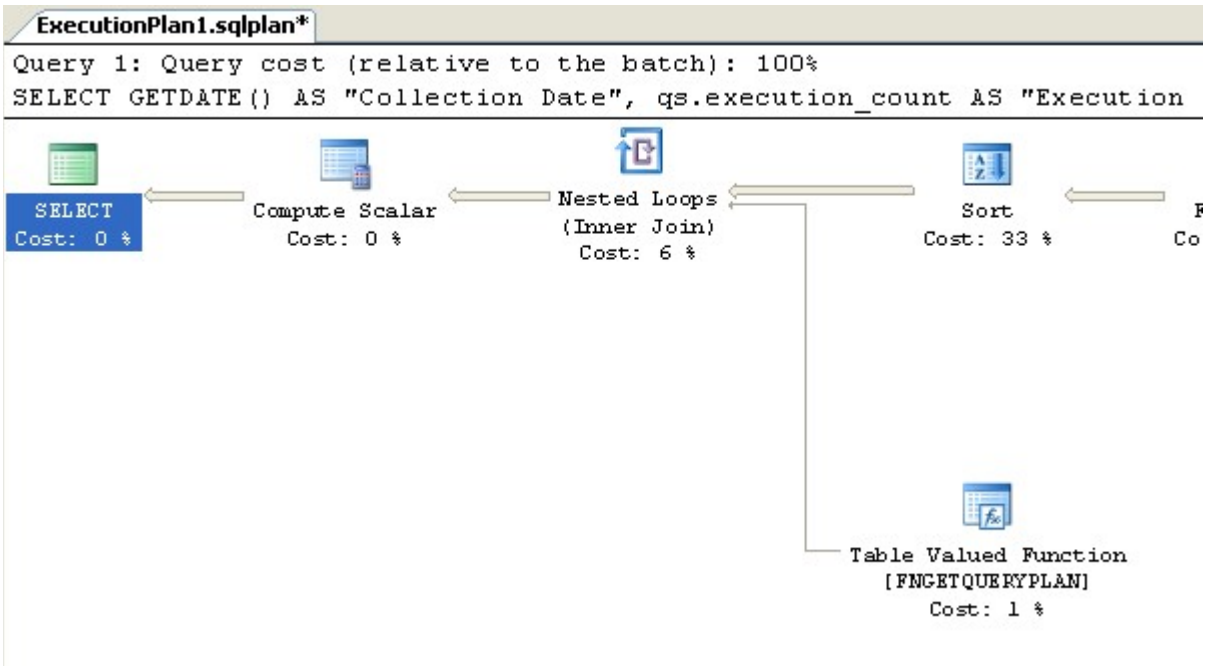
Collection Date	Execution Count	Query Text	DB Name
2012-01-20 09:44:33.030	4	SELECT TOP 20 GETDATE() AS "Collection Date", ...	NULL

Total CPU Time	Avg CPU Time (ms)	Total Physical Reads	Avg Physical Reads	Total Logical Reads
2288086	572021	0	0	2206

Avg Logical Reads	Total Logical Writes	Avg Logical Writes	Total Duration	Avg Duration (ms)
551	8	2	2536133	634033

Plan
<ShowPlanXML xmlns="http://schemas.microsoft.com/sql...

Also, if you click on the data in the "Plan" column it will display the execution plan in graphical format in a new tab.

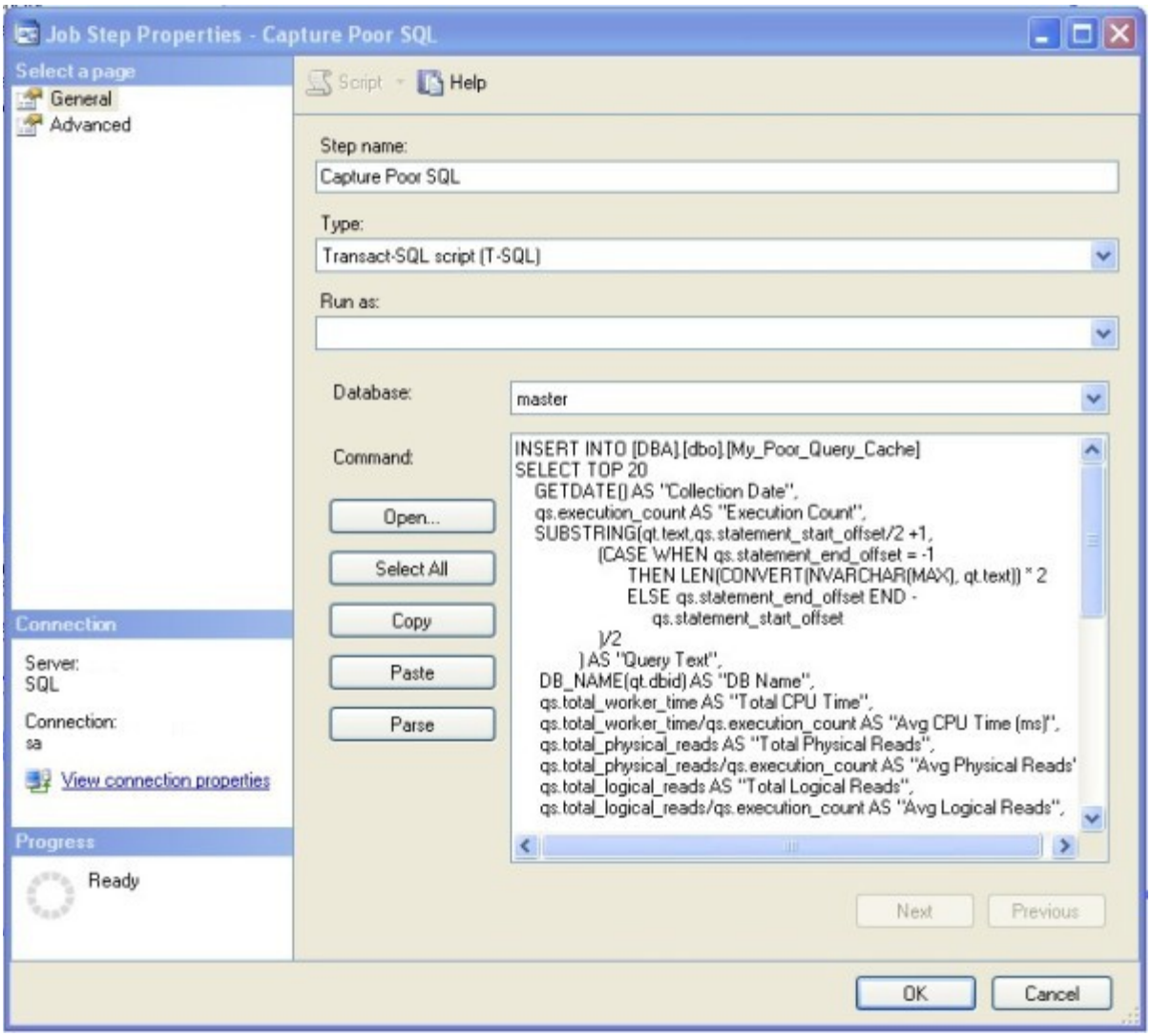


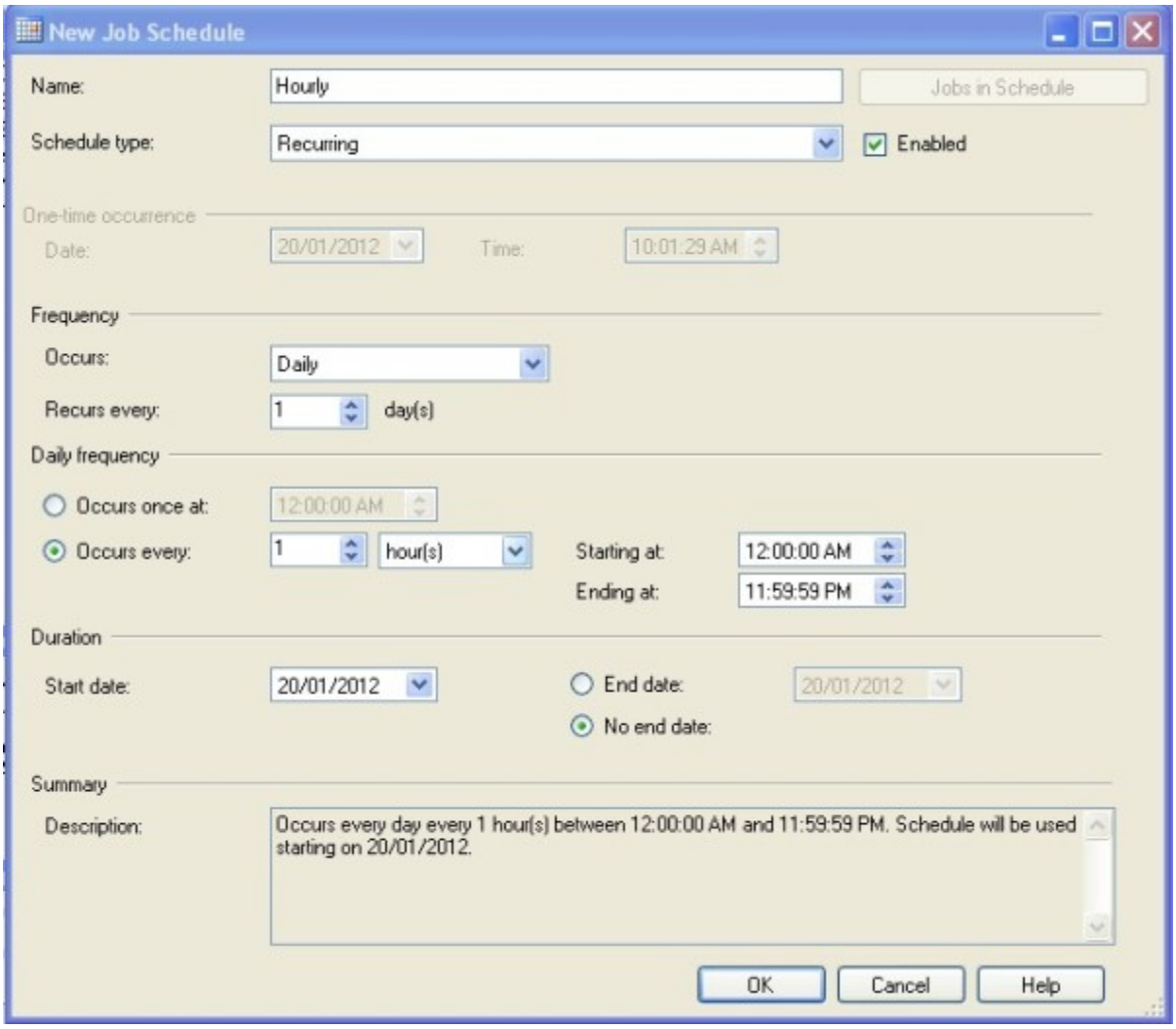
Now that we have a query to capture what we are looking for, we need somewhere to store the data. The following table definition can be used to store the output of the above query. We just have to add this line ahead of the query above to take care of inserting the result, "INSERT INTO [DBA].[dbo].[My_Poor_Query_Cache]".

```
CREATE TABLE [DBA].[dbo].[My_Poor_Query_Cache] (
 [Collection Date] [datetime] NOT NULL,
 [Execution Count] [bigint] NULL,
 [Query Text] [nvarchar](max) NULL,
 [DB Name] [sysname] NULL,
 [Total CPU Time] [bigint],
 [Avg CPU Time (ms)] [bigint] NULL,
 [Total Physical Reads] [bigint] NULL,
 [Avg Physical Reads] [bigint] NULL,
 [Total Logical Reads] [bigint] NULL,
 [Avg Logical Reads] [bigint] NULL,
 [Total Logical Writes] [bigint] NULL,
 [Avg Logical Writes] [bigint] NULL,
 [Total Duration] [bigint] NULL,
 [Avg Duration (ms)] [bigint] NULL,
 [Plan] [xml] NULL
) ON [PRIMARY]
GO
```

Finally, we'll use the SQL Server Agent to schedule this query to run. Your application and environment will determine how often you want to run this query. If queries stay in your SQL cache for a long period of time then this can be run fairly infrequently, however if the opposite is true they you may want to run this a little more often so any really poor SQL queries are not missed. Here are a few snapshots of the job I created. The T-SQL to create this job can be found

here.


Job Step Properties - Capture Poor SQL
Select a page
General
Advanced
Script
Help
Step name:
Capture Poor SQL
Type:
Transact-SQL script (T-SQL)
Run as:
Database:
master
Command:
INSERT INTO [DBA].[dbo].[My_Poor_Query_Cache]
SELECT TOP 20
GETDATE() AS "Collection Date",
qs.execution_count AS "Execution Count",
SUBSTRING(qt.text,qs.statement_start_offset/2 +1,
(CASE WHEN qs.statement_end_offset = -1
THEN LEN(CONVERT(NVARCHAR(MAX), qt.text)) * 2
ELSE qs.statement_end_offset END -
qs.statement_start_offset
)/2
) AS "Query Text",
DB_NAME(qt.dbid) AS "DB Name",
qs.total_worker_time AS "Total CPU Time",
qs.total_worker_time/qs.execution_count AS "Avg CPU Time (ms)",
qs.total_physical_reads AS "Total Physical Reads",
qs.total_physical_reads/qs.execution_count AS "Avg Physical Reads"
qs.total_logical_reads AS "Total Logical Reads",
qs.total_logical_reads/qs.execution_count AS "Avg Logical Reads",
Open...
Select All
Copy
Paste
Parse
Connection
Server:
SQL
Connection:
sa
View connection properties
Progress
Ready
Next
Previous
OK
Cancel



That's it. Now, whenever you have spare time you can query this table and start tuning.

SQL Server Query Execution Plans in SQL Server Management Studio

By: Tim Ford | Read Comments (10) | Related Tips: 1 | 2 | 3 | 4 | 5 | 6 | More > Query Plans

Problem

First, you click the 'New Query' button on the top left of SQL Server Management Studio to have a new query window to execute a query. Second, you click the 'Execute' icon on the toolbar or press the F5 key to process a query. Third, SQL Server does some processing behind the scenes. Fourth, results are returned. These are the four steps to retrieving information from Microsoft SQL Server. Three of which are pretty straightforward and happen in the light of day. You see the commands, parameters, object names, keywords and the like being laid down on the display in front of you when entering T-SQL commands. You are the one clicking the Execute or F5 key. You see the results as they are returned. Is there not a way to see how SQL Server goes about taking the gibberish of T-SQL and converting it to understandable rows of data? Of course there is. It's the graphical execution plan!

Solution

The graphical execution plan is just that. It is a tool that presents in images and text, the process by which SQL Server steps through the query, parsing, hashing, and building the results set using the information it has available (statistics, indexes, and the raw data). There are actually two other flavors of execution plans that we will not be discussing at this time: text execution plan and an xml execution plan. We will save those for another tip as it is necessary to have a good understanding of the graphical execution plan before moving on to those others that afford more detail.

There are two types of graphical execution plans: the estimated execution plan and the actual execution plan. They are precisely what the names imply (yes, a rarity in technology). The estimated execution plan is an estimate based upon the query optimizer on what it expects to occur when executing the query whereas the actual execution plan show what did actually occur when generating the results. For the purpose of this tip I'll be showing you how to run either graphical execution plan, but once that is presented the discussion will turn towards reading the basic graphical execution plan and will be agnostic as to whether it is the actual or estimated plan.

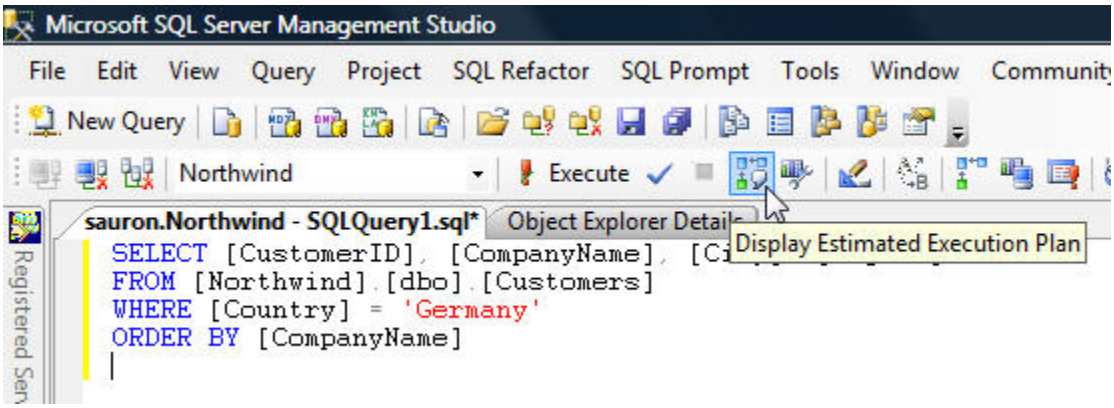
Presenting An Execution Plan

For the purpose of this tip we will be using the Northwind database. I've established a

connection to my test instance and have opened a new query to the Northwind database. The query is below and we will use this throughout this tip:

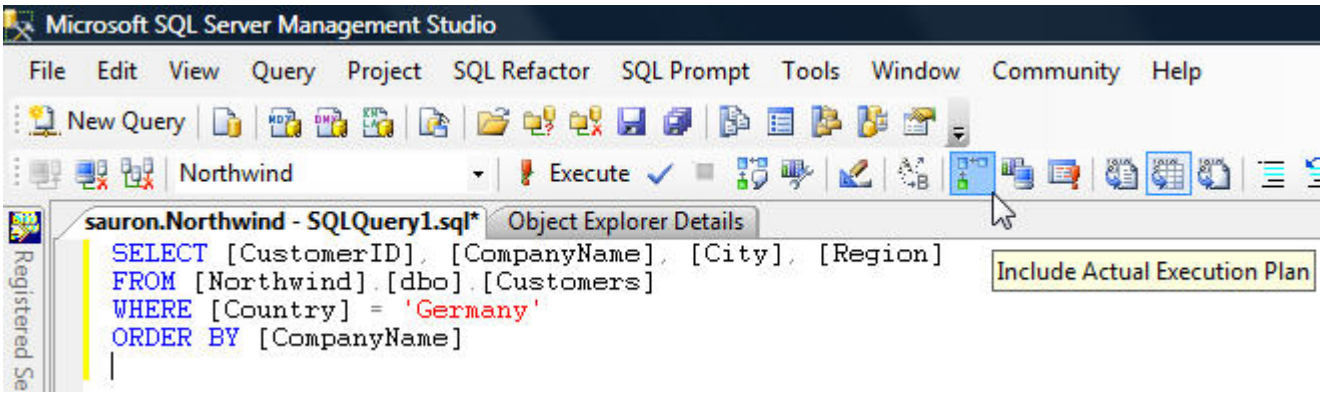
```
SELECT [CustomerID], [CompanyName], [City], [Region]
FROM [Northwind].[dbo].[Customers]
WHERE [Country] = 'Germany'
ORDER BY [CompanyName]
```

The estimated execution plan is engaged from the standard toolbar in SQL Server Management Studio as is highlighted below:



You can also use the Ctrl+L hotkey, right click the query window and select 'Display Estimated Execution Plan', or select 'Query/Display Estimated Execution Plan' from the SSMS menu bar to accomplish the same task.

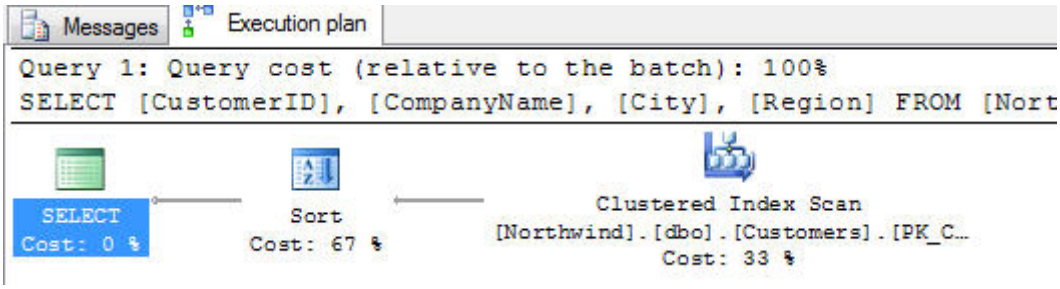
Presenting the Actual Execution Plans is slightly different, more so in behavior than in the functionality that triggers presentation of the query plan. Just as with Estimated Execution Plans there is a button in the SSMS application, you can right click the query and select 'Include Actual Execution Plan' from the pop-up window or do likewise from the menu bar; there is also a hotkey for Actual Execution Plans (in this case Ctrl+K). Enabling the Actual Execution Plan is a behavioral toggle for the SSMS application. This means that once you click the associated button, each time you execute any query you create in this or any query tab within SSMS the Actual Execution Plan for that query will be displayed. Clicking the button a second time turns off the behavior. The assigned button is shown below:



In the case of such a simple query as this one, both the actual and estimated query plans are identical. As a matter of fact, from this point, we will ignore whether the query plan we're observing is the Actual Execution Plan or the Estimated Execution Plan.

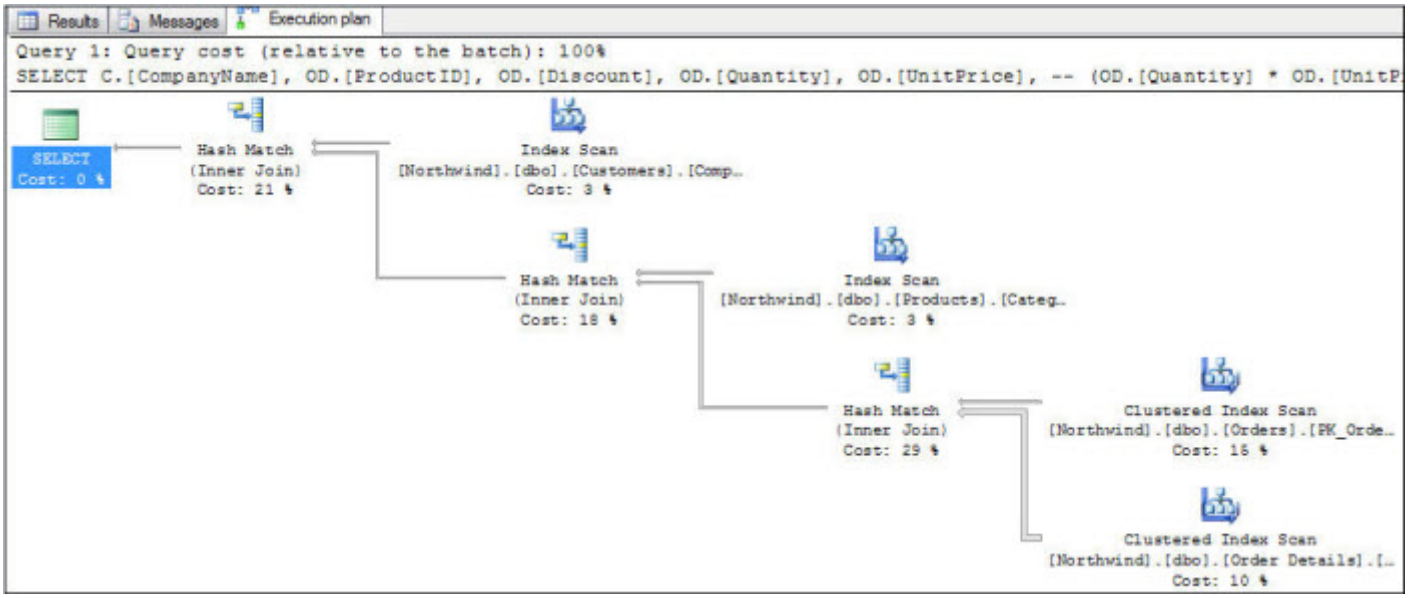
Reading the Graphical Execution Plan

Let's look at the execution plan for the aforementioned query and we'll begin the discussion on how to read such a plan. It is slightly nonsensical to those of us in Western culture and you'll see why in a second.



So why would I call this nonsensical? Take a look at the direction the arrows point within the execution plan and you'll see why. You read a graphical execution plan right-to-left. In future articles in this series you'll also see that in truth we read them right-to-left and top-to-bottom. For those of you who like to jump ahead to the good stuff in books I give you the following example of what you'll observe in terms of a more-complex execution plan similar to what we will digest in detail later in the series:

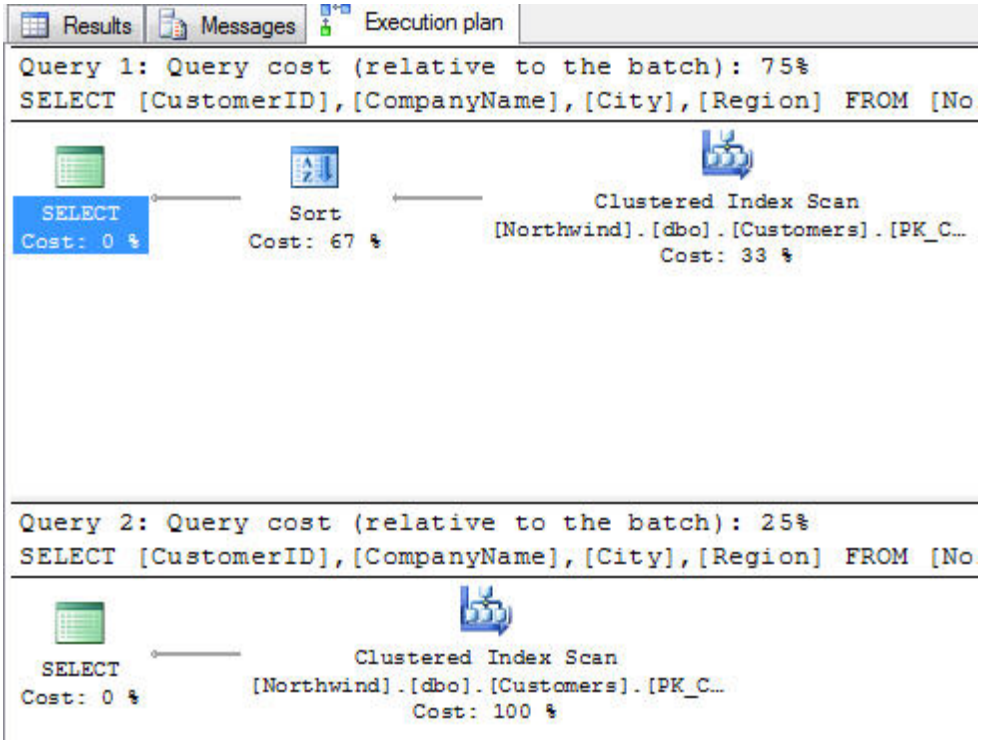
```
SELECT C.[CompanyName], OD.[ProductID], OD.[Discount],
  OD.[Quantity], OD.[UnitPrice],
  O.[OrderDate], O.[RequiredDate], O.[ShippedDate]
FROM Orders O
  INNER JOIN [dbo].[Order Details] OD
     ON O.[OrderID] = OD.[OrderID]
  INNER JOIN [dbo].[Products] P
     ON OD.[ProductID] = P.[ProductID]
  INNER JOIN [dbo].[Customers] C
     ON C.[CustomerID] = O.[CustomerID]
WHERE O.[ShippedDate] > O.[RequiredDate]
```



The specifics of this more advanced query are unimportant at this time; all that is important is that you understand that the general reading of the query plan is, as I have stated right-to-left and top-to-bottom. This more advanced query also highlights another notable display behavior and that is the size of the arrow used to denote the flow of data from one process to the next. The thickness of the arrow correlates to the number of rows flowing between the steps. But I digress...Let's get back to the first execution plan, shall we? Besides the flow of information and arrow symbolism you'll also notice that each step/process has an associated cost. This is a percentage of cost for the step compared to the total cost of all steps in the query plan. Rounding is obviously involved, so when you do see 0% for a cost you need to understand that some time was incurred to complete the step. Nothing, I repeat NOTHING is free in a SQL query. There is yet another cost presented initially in the query plan and that is located in the header for the graphical query plan. This cost is a factor if you run more than a single T-SQL statement in a batch. Each statement will have an associated query plan, and this metric displays the cost for each statement when compared to the total for all statements run in the batch. As a final task in this tip let's take a quick look at this behavior. What happens when we execute the following two statements in the same execution batch?

```
SELECT [CustomerID], [CompanyName], [City], [Region]
FROM [Northwind].[dbo].[Customers]
WHERE [Country] = 'Germany'
ORDER BY [CompanyName]

SELECT [CustomerID], [CompanyName], [City], [Region]
FROM [Northwind].[dbo].[Customers]
WHERE [Country] = 'Germany'
```



The statements were identical, other than the ORDER BY clause was omitted from the second statement. When you compare the two statements, the metrics presented show that the first query consumes 75% of the total execution time for the batch, whereas the second query consumes only 25% of the batch's total cost. This is a relative cost metric, as labeled appropriately.

To summarize, graphical execution plans:

- There are a variety of methods for displaying or triggering graphical execution plans
- You read them right-to-left and top-to-bottom
- The arrows denote not just the direction of data travel, but also (comparatively) amount of data rows being transferred from step-to-step in the execution process
- Costs are displayed for each step, relative to the total cost of the query plan
- The cost for the statement can be compared to other statements run in the same batch

In subsequent tips we will explore many of the various types of processes you'll see presented as individual steps in the query execution plan. We will also explore the additional wealth of information presented as pop-ups and properties of each step within the SQL Server Management Studio.

How to read SQL Server graphical query execution plans

By: Tim Ford | Read Comments (5) | Related Tips: 1 | 2 | 3 | 4 | 5 | 6 | More > Query Plans

Problem

In the previous tip in this series, I gave you an initial look into how to launch and read a graphical query execution plan. We're going to dive deeper into the different opportunities for information sources *within* the graphical execution plan itself. Yes, you read that right. There are other sources of information in the graphical execution plan that are not readily apparent at first blush. This tip will focus on Tooltips and the next on the Properties window.

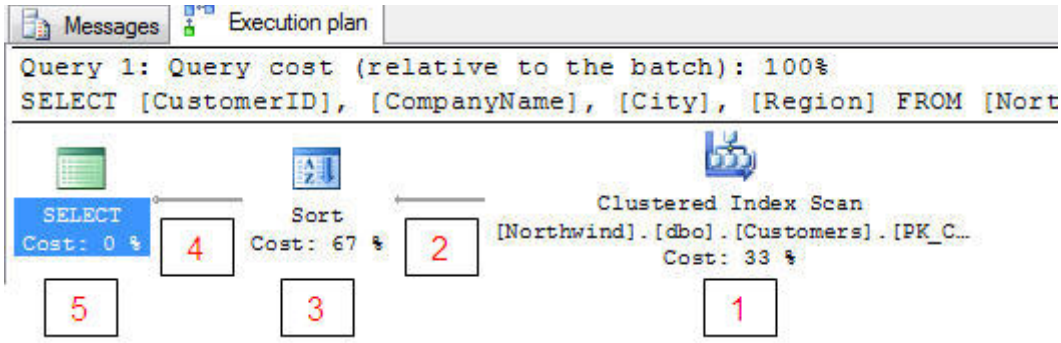
Solution

Those of you who have been reading my articles for the past two years have heard me state the values of *being able to read Microsoft*. What I mean is that Microsoft as a software development company spends many hours ensuring that their products have similar graphical user interface (GUI) design and behavior. Frequent users of their products be they accountants using Excel or DBAs using SQL Server Management Studio know that if they were to select *Tools* from the menu bar in their application that they could drill down into *Options* and then make modifications to their instance of the product. They know that if they right click in the application they will get a pop-up window with additional options. They also know about the tool Tooltips that occasionally pop-up when you hover over a toolbar button or area of the GUI long enough; those little yellow boxes with additional read-only information about the specific aspect of the GUI that they are hovering the mouse over at that point in time.

Just like the many other Microsoft products, SQL Server Management Studio also has those tooltips. The graphical query execution plan takes it to a new level, with monster-sized tooltips as you will soon see.

Remember that first query and graphical execution plan I presented in the earlier tip in the series? If not, here it is below. I will continue to use it as the basis for our discussion. This is a very simple SELECT statement against the Northwind sample database in SQL Server 2005, with a filter and sort operation thrown in for good measure.

```
SELECT [CustomerID], [CompanyName], [City], [Region]
FROM [Northwind].[dbo].[Customers]
WHERE [Country] = 'Germany'
ORDER BY [CompanyName]
```

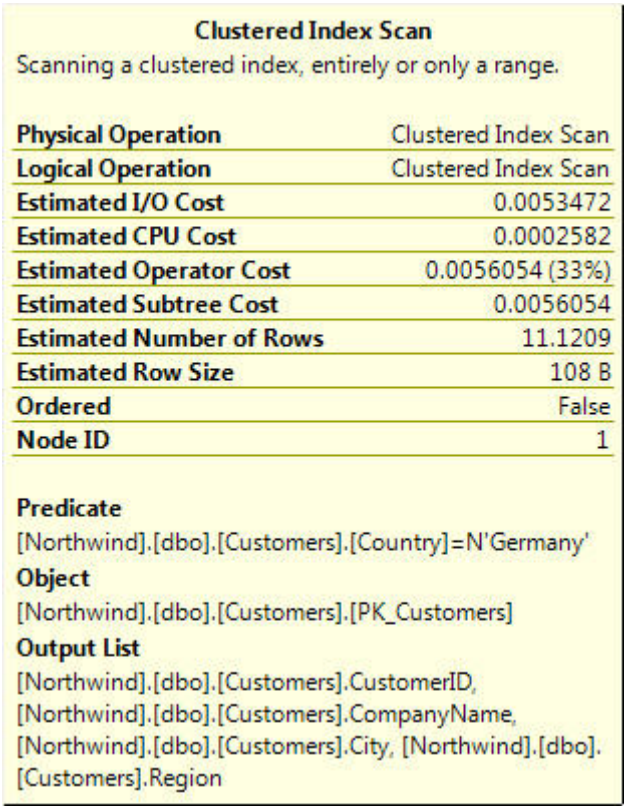


In the above estimated execution plan, I have added numbers 1 through 5 to help better explain each operation.

Let's look at the tooltips for each of the operations in this simple execution plan, working our way from right-to-left just as you would read the plan itself. You will see similarities between the tooltips for each operation, but you should note that there are quite a few differences as well. You also may be surprised to learn that even the arrows denoting data flow between the operator icons have associated tooltips.

So, if you hover over each item in the execution plan you will get different information. The following shows each set of information for these 5 parts of this execution plan.

1 - Clustered Index Scan Operator



Clustered Index Scan

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated I/O Cost	0.0053472
Estimated CPU Cost	0.0002582
Estimated Operator Cost	0.0056054 (33%)
Estimated Subtree Cost	0.0056054
Estimated Number of Rows	11.1209
Estimated Row Size	108 B
Ordered	False
Node ID	1

Predicate
[Northwind].[dbo].[Customers].[Country]=N'Germany'
Object
[Northwind].[dbo].[Customers].[PK_Customers]
Output List
[Northwind].[dbo].[Customers].CustomerID, [Northwind].[dbo].[Customers].CompanyName, [Northwind].[dbo].[Customers].City, [Northwind].[dbo].[Customers].Region

I want to take some time here to review each of the metrics offered in this, the first tooltip we've looked at so far. From here we will only focus on the new / different line items in subsequent tooltips for each operation. You'll see that the tooltip presents the type of operation as well as a standardized description of the operation initially. This is followed by (in the case of estimated execution plans) estimated operational metrics. If this was an *actual* execution plan you would have also seen the Actual Number of Rows involved in the operation after the Physical Operation and Logical Operation metrics.

- Physical Operation - the physical operation for this part of the execution plan, such as joins, seeks, scans...
- Logical Operation - the logical operation for this part of the execution plan
- Estimated I/O Cost - these are relative values used for presenting whether a given operation is I/O intensive. The Query Optimizer assigns these values during parsing and they serve only as a comparative tool to aid in determining where the costs of a given operation lie. The larger the value, the more cost-intensive the process.
- Estimated CPU Cost - these are relative values used for presenting whether a given operation is CPU intensive. The Query Optimizer assigns these values during parsing and they serve only as a comparative tool to aid in determining where the costs of a given operation lie. The larger the value, the more cost-intensive the process.
- Estimated Operator Cost - This is the summation of the I/O and CPU estimated costs. This is the value that is also presented under each operation icon in the graphical execution plan.
- Estimated Subtree Cost - This is the total of this operation's cost as well as all other

operations that preceded it in the query to this point.

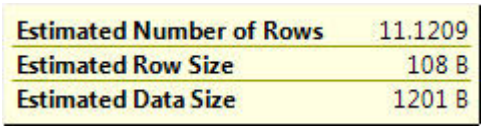
- Estimated Number of Rows - This value is derived based upon the statistics available to the Query Optimizer at the time the execution plan is drafted. The more current (and the larger the sampling size of the statistics) the more accurate this metric and others will be when compared to the actual data.
- Estimated Row Size - Also based upon the statistics available at the time of parsing, this value corresponds to how wide the Query Optimizer believes the affected rows to be. The same rule applies to statistics here as well as with the Estimated Number of Rows - the more current and descriptive the data set used to generate the statistics - the more accurate this value will be in comparison to the actual data. Good stats also lead to better (more accurate) decisions made by the Query Optimizer when actually processing your queries.
- Ordered - is a Boolean value signifying whether the rows are ordered in the operation.
- NodeID - Is the ordinal value associated with this particular operation in the query execution plan. Oh, and thank you for the confusion Microsoft for numbering the operations in a left-to-right fashion, even though we are to read the execution plans right-to-left. Greatly appreciated!

Other items available in actual execution plans

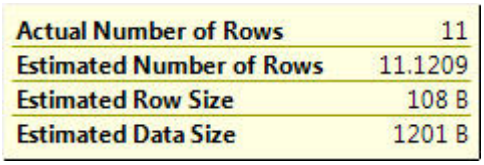
- Actual Number of Rows - The actual number of rows if the query was run.
- Actual Rebinds and Actual Rewinds - The topic of rebinds and rewinds is outside the scope of this tip. Suffice to say the value is incremented in different manners when a given operation process is initialized. A change in correlated parameters within a join would be noted as a rebind - the join would need to be re-evaluated, whereas a rewind count is incremented if these parameters are not changed and the existing join can be used for subsequent processing. What compounds the complexity of this subtopic is that not all operations present rebinds and rewinds to the Query Optimizer. I will point you many times to an excellent book by a good friend of mine and outstanding SQL Server Professional, Grant Fritchey. His book on the subject of query execution plans is a fantastic wealth of information and is widely available as either a hard copy or as a free digital download. He does a good job of explaining the topic of rebinds and rewinds. Any attempt I would make to do so would just be a re-hashing of the material he provides in his book.

These line items are then followed by the Predicate, Object, and Output List of columns for the operator. *Predicate* is the term used to describe the portion of a query used to filter, describe, or compare sets of data. In this case it the portion of the query that filters the results for just those rows where the country of interest is equivalent to 'Germany'. The object used as a tool to accomplish this task for this specific operator was the primary key on the Customers table. The output list is just that, the list of columns that are output from the process being described: CustomerID, CompanyName, City, and Region (the columns that are to be returned in our SELECT statement.

2- Data Flow Arrow: Clustered Index Scan Operator to Sort Operator



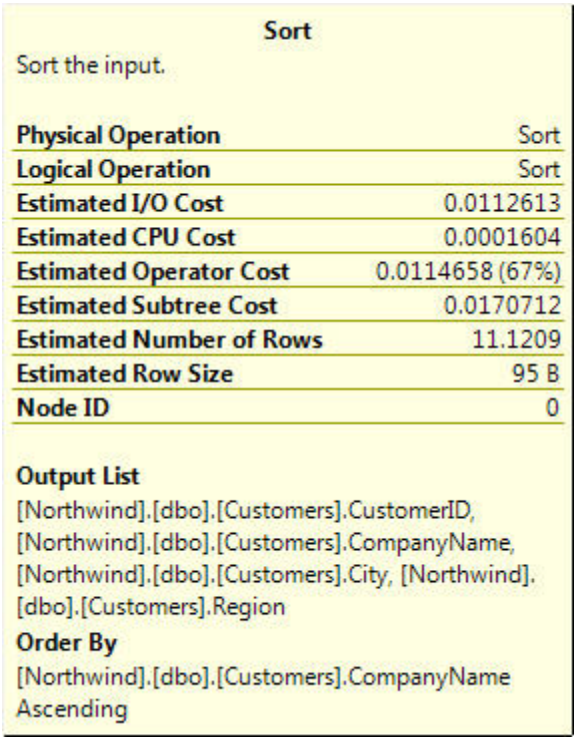
Can you tell by just this tooltip whether we're reviewing an actual or an estimated graphical execution plan? It's not as simple as you would expect. Both types of graphical execution plans provide these rows in their tooltip. However, the actual query plan also includes the Actual Number of Rows:



The tooltips associated with the data flow arrows are quite simple, they provide information relating to the estimated (or actual) data moving between operations in a query.

---

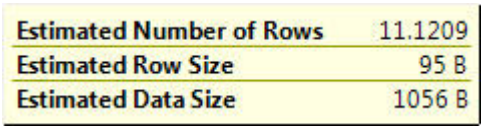
3 -Sort Operator



The tooltip for the Sort operator and Clustered Index Scan operator are identical in the metrics they capture. Obviously the values for these metrics differ. You'll see that the Estimated Subtree Cost has now been incremented by the cost of this operation as well as all preceding operations.

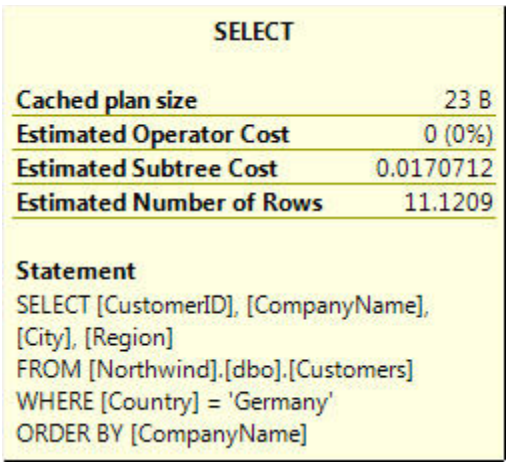
---

4 - Data Flow Arrow: Sort Operator to SELECT Operator



Identical in content (but not necessarily values) we next encounter the data flow arrow between the Sort and SELECT operations. We're almost done!

5- SELECT Operator



Take note that the SELECT operator's tooltip is drastically different from the other operator tooltips we've seen so far - the DML (Data Modification Language actions such as INSERT UPDATE DELETE) will also differ compared to the other operators we've seen to date as well as the SELECT operator presented here. We will look at the other DML operator's tooltips as they come into the discussion in future tips. In the case of the SELECT operator we have a new line item: Cached plan size. Cached plan size denotes the amount of memory this query plan is consuming in the stored procedure cache. This value is useful if you are troubleshooting memory issues specifically-related to cache performance.

There you have it. Our initial dive into reading the wealth of information presented in the tooltips for the graphical query execution plans in Microsoft SQL Server. These tooltips provide us with substantially more information than the graphic portion of the execution plan. From costing information to amount of actual data affected at each step of the process, the tooltips give us a greater insight into why and how the Query Optimizer responds to our T/SQL code. Our next tip in the series will focus on the Properties window available to us in SQL Server Management Studio as it pertains to execution plans.

Last Update: 11/4/2009