Feb 20, 2009

**Finding the Worst Performing T-SQL Statements on an Instance**

**By** [**Gregory A. Larsen**](http://www.databasejournal.com/feedback.php/http:/www.databasejournal.com/features/mssql/article.php/3802936/Finding-the-Worst-Performing-T-SQL-Statements-on-an-Instance.htm)

One of the key things to consider when building a successful application is to make sure the application performance meets the customer’s expectation. As DBAs and Developers, we need to do everything we can to ensure that the T-SQL code to support an application is running as efficiently as it can. You should not wait for customers to complain, but instead you should be proactively monitoring your applications to make sure the code that is being run is as efficient as possible. When code is found that uses excessive CPU, I/O or has a long duration you need to analyze the code to determine if the performance can be improved. In this article, I will introduce you to some Dynamic Management Views (DMVs) and Dynamic Management Functions (DMFs) that will help you monitor your application’s T-SQL performance. In addition, I will provide you with a Stored Procedure (SP) that will utilize the DMVs and DMFs to produce a report that will allow you to quickly identify poorly performing T-SQL statements, at either the instance, or database level.

**Using DMVs and DMFs to Identify Poorly Performing T-SQL**

When Microsoft introduced SQL Server 2005 they provided DBAs and Developer with Dynamic Management Views (DMVs) and Dynamic Management Functions (DMFs). These new views provide you with a wealth of information about how your database instance is running. Using these new objects allows you can look at the internal workings of SQL Server with simple SELECT statements. There are four different DMVs and DMFs that when used together allow you to peek under the covers of the database engine and return resources usage statistics for T-SQL statements that have executed on your instance. Here is a list of those DMVs and DMFs: sys.dm\_exec\_query\_stats, sys.dm\_exec\_cached\_plans, sys.dm\_exec\_sql\_text, and sys.dm\_exec\_plan\_attributes. The sys.dm\_exec\_query\_stats DMV provides aggregated statistics like I/O, CPU and Elapsed Time for statements within a cached plan. The sys.dm\_exec\_cached\_plans allows you the ability to identify how often a cached plan has been used. The T-SQL text of each statement in a cached plan can be found by using the sys.dm\_exec\_sql\_text DMF. Lastly, the sys.dm\_exec\_plan\_attributes DMF can be used to identify different attributes of a plan, like DBID.

These DMV’s and DMFs get their information from within memory. SQL Server keeps statistics in memory from the time SQL Server starts up until it shuts down. If SQL Server has not been up very long the statistics these DMVs and DMFs return might not be a very representative sample of normal types of T-SQL statement executed on an instance, as it might if SQL Server has been up for a number of days, or weeks. Also worth noting is that T-SQL execution statistics are only available for the statements that still have their execution plans in memory. If a statement is executed and then its execution plan is overwritten, then its associated statistics are flushed when the execution plan is overwritten.

To further understand how to use these DMVs and DMFs I will go through a few examples. Let’s first review this code:

SELECT \*

FROM sys.dm\_exec\_query\_stats qs

JOIN sys.dm\_exec\_cached\_plans cp ON qs.plan\_handle = cp.plan\_handle

CROSS APPLY sys.dm\_exec\_sql\_text(qs.plan\_handle) st

If you run this code against your SQL Server instance, you will be able to see statistics that sys.dm\_exec\_query\_stats DMV is able to show out of the cached plans like:

* How many times a give execution plan has been executed.
* The total CPU that a given T-SQL statement has consumed for all executions.
* The total number of physical read, logical writes and logical read a given T-SQL statement has used for all executions.
* The total amount of elapsed time it has taken to execute a given T-SQL statement.

These statistics can help you identify which T-SQL statements are performing poorly based on CPU, I/O, or elapsed time.

By joining the sys.dm\_exec\_query\_stats DMV with the sys.dm\_exec\_cached\_plans DMV in the above query, you are able to identify the type of plan being used like: Proc, Adhoc, and Prepared. This information will help you identify which T-SQL statements are coming from Stored Procedures and those that are submitted dynamically or directly from some applications.

By using the CROSS APPLY operator in conjunction with the sys.dm\_exec\_sql\_text DMF you can return the T-SQL batch text associated with an execution plan. Since a batch can contain multiple T-SQL statements, you can use the offset columns from the sys.dm\_exec\_query\_stats to identify the actual T-SQL statement within the batch with which the statistics are associated.

The above code shows you all the statistics that a SQL Server instance is compiling across all databases. When you are tuning a specific application, you might want to narrow down the statistics being displayed to a specific database. You can do this by adding the sys.dm\_exec\_plan\_attributes DMF to the above T-SQL code like so:

SELECT \*

FROM sys.dm\_exec\_query\_stats qs

JOIN sys.dm\_exec\_cached\_plans cp ON qs.plan\_handle = cp.plan\_handle

CROSS APPLY sys.dm\_exec\_sql\_text(qs.plan\_handle) st

OUTER APPLY sys.dm\_exec\_plan\_attributes(qs.plan\_handle) pa

WHERE attribute = 'dbid'

AND DB\_NAME(CAST(pa.value AS int)) = 'AdventureWorks'

Here I have used the OUTER APPLY operator along with a WHERE clause to narrow down the statistics to only those that are associated with statements from within the “AdventureWorks” database. In reality, the “dbid” attribute from the sys.dm\_exec\_plan\_attribute DMF contains the DBID in which the session was connected when the T-SQL statement was executed. If you are using three part naming conventions for your database objects then the plan attribute dbid value might not be the actual dbid against which the statements associated with the cached plan was run.

Now that you have an understanding of the DMF’s and DMVs that can be used to obtain T-SQL execution statistics let me show you how I used the information from these management views to develop a parameter driven SP.

**usp\_Worst\_TSQL**

My SP is call "usp\_Worst\_TSQL". This SP produces a report of worst T-SQL statements, by instance or database. Note that this SP uses a new function "OBJECT\_SCHEMA\_NAME" that was introduced with SQL Server 2005 SP2. If you are on an earlier version of SQL Server 2005 you will need to modify the code in the SP below. By passing optional parameters to this SP, you can focus in on different types of performance problems, like CPU, I/O or duration. In addition, you can identify the database and the number of poorly performing T-SQL statements you want returned. This SP allows DBAs a quick way to return query statistics using the above mentioned DMV’s and DMF’s without having to remember how to join these views and functions. Here is the code for my SP:

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

CREATE PROC [dbo].[usp\_Worst\_TSQL]

/\*

Written by: Gregory A. Larsen

Copyright © 2008 Gregory A. Larsen. All rights reserved.

Name: usp\_Worst\_TSQL

Description: This stored procedure displays the top worst performing queries based on CPU, Execution Count,

I/O and Elapsed\_Time as identified using DMV information. This can be display the worst

performing queries from an instance, or database perspective. The number of records shown,

the database, and the sort order are identified by passing pararmeters.

Parameters: There are three different parameters that can be passed to this procedures: @DBNAME, @COUNT

and @ORDERBY. The @DBNAME is used to constraint the output to a specific database. If

when calling this SP this parameter is set to a specific database name then only statements

that are associated with that database will be displayed. If the @DBNAME parameter is not set

then this SP will return rows associated with any database. The @COUNT parameter allows you

to control the number of rows returned by this SP. If this parameter is used then only the

TOP x rows, where x is equal to @COUNT will be returned, based on the @ORDERBY parameter.

The @ORDERBY parameter identifies the sort order of the rows returned in descending order.

This @ORDERBY parameters supports the following type: CPU, AE, TE, EC or AIO, TIO, ALR, TLR, ALW, TLW, APR, and TPR

where "ACPU" represents Average CPU Usage

"TCPU" represents Total CPU usage

"AE" represents Average Elapsed Time

"TE" represents Total Elapsed Time

"EC" represents Execution Count

"AIO" represents Average IOs

"TIO" represents Total IOs

"ALR" represents Average Logical Reads

"TLR" represents Total Logical Reads

"ALW" represents Average Logical Writes

"TLW" represents Total Logical Writes

"APR" represents Average Physical Reads

"TPR" represents Total Physical Read

Typical execution calls

Top 6 statements in the AdventureWorks database base on Average CPU Usage:

EXEC usp\_Worst\_TSQL @DBNAME='AdventureWorks',@COUNT=6,@ORDERBY='ACPU';

Top 100 statements order by Average IO

EXEC usp\_Worst\_TSQL @COUNT=100,@ORDERBY='ALR';

Show top all statements by Average IO

EXEC usp\_Worst\_TSQL;

\*/

(@DBNAME VARCHAR(128) = '<not supplied>'

,@COUNT INT = 999999999

,@ORDERBY VARCHAR(4) = 'AIO')

AS

-- Check for valid @ORDERBY parameter

IF ((SELECT CASE WHEN

@ORDERBY in ('ACPU','TCPU','AE','TE','EC','AIO','TIO','ALR','TLR','ALW','TLW','APR','TPR')

THEN 1 ELSE 0 END) = 0)

BEGIN

-- abort if invalid @ORDERBY parameter entered

RAISERROR('@ORDERBY parameter not APCU, TCPU, AE, TE, EC, AIO, TIO, ALR, TLR, ALW, TLW, APR or TPR',11,1)

RETURN

END

SELECT TOP (@COUNT)

COALESCE(DB\_NAME(st.dbid),

DB\_NAME(CAST(pa.value AS INT))+'\*',

'Resource') AS [Database Name]

-- find the offset of the actual statement being executed

,SUBSTRING(text,

CASE WHEN statement\_start\_offset = 0

OR statement\_start\_offset IS NULL

THEN 1

ELSE statement\_start\_offset/2 + 1 END,

CASE WHEN statement\_end\_offset = 0

OR statement\_end\_offset = -1

OR statement\_end\_offset IS NULL

THEN LEN(text)

ELSE statement\_end\_offset/2 END -

CASE WHEN statement\_start\_offset = 0

OR statement\_start\_offset IS NULL

THEN 1

ELSE statement\_start\_offset/2 END + 1

) AS [Statement]

,OBJECT\_SCHEMA\_NAME(st.objectid,dbid) [Schema Name]

,OBJECT\_NAME(st.objectid,dbid) [Object Name]

,objtype [Cached Plan objtype]

,execution\_count [Execution Count]

,(total\_logical\_reads + total\_logical\_writes + total\_physical\_reads )/execution\_count [Average IOs]

,total\_logical\_reads + total\_logical\_writes + total\_physical\_reads [Total IOs]

,total\_logical\_reads/execution\_count [Avg Logical Reads]

,total\_logical\_reads [Total Logical Reads]

,total\_logical\_writes/execution\_count [Avg Logical Writes]

,total\_logical\_writes [Total Logical Writes]

,total\_physical\_reads/execution\_count [Avg Physical Reads]

,total\_physical\_reads [Total Physical Reads]

,total\_worker\_time / execution\_count [Avg CPU]

,total\_worker\_time [Total CPU]

,total\_elapsed\_time / execution\_count [Avg Elapsed Time]

,total\_elapsed\_time [Total Elasped Time]

,last\_execution\_time [Last Execution Time]

FROM sys.dm\_exec\_query\_stats qs

JOIN sys.dm\_exec\_cached\_plans cp ON qs.plan\_handle = cp.plan\_handle

CROSS APPLY sys.dm\_exec\_sql\_text(qs.plan\_handle) st

OUTER APPLY sys.dm\_exec\_plan\_attributes(qs.plan\_handle) pa

WHERE attribute = 'dbid' AND

CASE when @DBNAME = '<not supplied>' THEN '<not supplied>'

ELSE COALESCE(DB\_NAME(st.dbid),

DB\_NAME(CAST(pa.value AS INT)) + '\*',

'Resource') END

IN (RTRIM(@DBNAME),RTRIM(@DBNAME) + '\*')

ORDER BY CASE

WHEN @ORDERBY = 'ACPU' THEN total\_worker\_time / execution\_count

WHEN @ORDERBY = 'TCPU' THEN total\_worker\_time

WHEN @ORDERBY = 'AE' THEN total\_elapsed\_time / execution\_count

WHEN @ORDERBY = 'TE' THEN total\_elapsed\_time

WHEN @ORDERBY = 'EC' THEN execution\_count

WHEN @ORDERBY = 'AIO' THEN (total\_logical\_reads + total\_logical\_writes + total\_physical\_reads) / execution\_count

WHEN @ORDERBY = 'TIO' THEN total\_logical\_reads + total\_logical\_writes + total\_physical\_reads

WHEN @ORDERBY = 'ALR' THEN total\_logical\_reads / execution\_count

WHEN @ORDERBY = 'TLR' THEN total\_logical\_reads

WHEN @ORDERBY = 'ALW' THEN total\_logical\_writes / execution\_count

WHEN @ORDERBY = 'TLW' THEN total\_logical\_writes

WHEN @ORDERBY = 'APR' THEN total\_physical\_reads / execution\_count

WHEN @ORDERBY = 'TPR' THEN total\_physical\_reads

END DESC

As you can see this SP supports three different parameters. The first parameter @DBNAME allows you to identify the database you want to look at for the worst performing T-SQL statements. If this parameter is not populated, the SP returns information for all databases. The second parameter, @COUNT, identifies the number of worst performing T-SQL statements you want to return. The default number is 999999999, which essentially identifies all the statistics SQL Sever is retaining in memory. The last parameter @ORDERBY allows you to return the statistics sorted based on one of the DMV statistics columns. Look at the comments in the code above to determine what are acceptable values for the @ORDERBY clause.

Here is a typical call to this SP:

EXEC usp\_Worst\_TSQL

@DBNAME='AdventureWorks',

@COUNT=5,

@ORDERBY='AIO';

This particular call will return the top five worst performing T-SQL statements in the AdventureWorks database based on Average I/O used. You could use this call to narrow in on statements within the AdventureWorks database that were using an excessive number of I/O’s on average.

If you wanted to return the top 100 worse performing commands based on average elapsed time across all your databases you would call this SP with the following code:

EXEC usp\_Worst\_TSQL

@COUNT=100,

@ORDERBY='AE';

When you are doing a performance review of an application or an instance this SP becomes a good tool to use. By using this SP, you can quickly determine those queries that are consuming the most resources. Using this kind of information allows you to quickly narrow in on the T-SQL statements or stored procedures that you should focus your attention on if you want to improve performance.

**Low Impact Performance Monitoring Tool**

There is real value in knowing which queries to optimize when you are doing a performance tune-up of your application. If you want to identify the poorly performing T-SQL statements or SPs that are using the most resources then the DMVs and DMFs identified here is one option for providing this information. Having a simple tool, like the SP example above will allow you a quick method of determining which commands are the worst performing based on CPU, I/O or elapsed time. Next time you have a need to look into the query execution performance consider using the DMV’s and DMF’s I mentioned for a low impact performance monitoring tool solution.