5 Monitoring Queries for SQL Server

01 June 2016

*by* [*Dennes Torres*](https://www.simple-talk.com/author/dennes-torres/)

FROM: <https://www.simple-talk.com/sql/database-administration/5-monitoring-queries-for-sql-server/>

Every DBA squirrels away favourite queries for monitoring SQL Server. Nowadays many of these are too complex to keep in your head. Dennes describes how he uses T-SQL queries for solving problems, whether it involves fixing the problems of missing indexes, preventing unrestrained autogrowth, avoiding index fragmentation, checking whether jobs have failed  or avoiding memory stress conditions.

Every DBA needs a toolset full of queries to monitor SQL Server. Some queries to monitor the server are complex and take time to be built, so the DBA needs to have these queries readily to hand, maybe on a USB thumb-drive, but for use at a moment’s notice.

In this article, I’m going to introduce the five important queries that I use to monitor SQL Server.

**Missing index**

In a perfect world, you, the DBA, would have enough time to analyze all the queries that an application makes to SQL Server and ensure that the correct indexes are there in place to support those queries. In a perfect world, no developers could send new queries to the production environment before the DBA has enough time to analyze the queries.

We are not in a perfect world. Applications often go live without enough time for the DBA to analyze all the queries. SQL Server has a feature to work around these situations. The query optimizer can identify, during the query compilation, whether the query could have a better result with a new index. In this case, the query optimizer registers this information in some system objects. We can query DMVs to find out the missing index for our queries and optimize our database environment.

The DMVs are:

* **sys.dm\_db\_missing\_index\_details**: This DMV has details about the keys that we can use to create new indexes.
* **sys.dm\_db\_misssing\_index\_group\_stats**: This DMV has details about how the missing index would be used were it to exist. It tells you how many seeks and scans would be done over the missing index.
* **sys.dm\_db\_missing\_index\_groups**: This DMV has keys to relate the two results: provided by the **sys.dm\_db\_missing\_index\_details** and **sys.dm\_db\_missing\_index\_group\_stats** DMVs.

Before get a result from these DMVs, we need to understand how the missing index keys are stored in **sys.dm\_db\_missing\_index\_details**. There are three fields containing key information, they are:

* **Equality\_Columns:** This field contains information about all the keys that will be used by predicates with the equal sign ("="). These keys should be included first in a composite index.
* **Inequality\_Columns:** This field contains information about all the keys that will be used by predicates without the equal sign, it could be any comparison as greater than or lower than. These keys should be included last in composite indexes.
* **Included\_Columns:** This field contains columns that shouldn't be part of the index keys, but should be used as included columns, with the INCLUDE statement in the CREATE INDEX instruction.

These DMVs will capture both the whole range of queries, ranging from those queries with high use and impact through to queries that are used only once and that will never executed again. We can use the information in **sys.dm\_db\_missing\_index\_group\_stats**to identify the importance of each missing index information.

The **sys.dm\_db\_missing\_index\_group\_stats** has these particularly interesting fields:

* **user\_seeks:** This column tells us how many seek operations would be done over the missing index.
* **user\_scans:** This column tells us how many scan operations would be done over the missing index.
* **avg\_total\_user\_cost:** This column has an average cost for all the queries that would have used the missing index. The "cost" is a ‘unitless’ value in an arbitrary currency, calculated by the query optimizer, but it is useful to know.
* **avg\_user\_impact:** This column has an average percentage by which query cost could drop if we create this missing index.

Now we have enough information to create a query to show us the missing index in the order of the impact their creation will have:

SELECT TOP 20

ROUND(s.avg\_total\_user\_cost \*

s.avg\_user\_impact

\* (s.user\_seeks + s.user\_scans),0)

AS [Total Cost]

,d.[statement] AS [Table Name]

,equality\_columns

,inequality\_columns

,included\_columns

FROM sys.dm\_db\_missing\_index\_groups g

INNER JOIN sys.dm\_db\_missing\_index\_group\_stats s

ON s.group\_handle = g.index\_group\_handle

INNER JOIN sys.dm\_db\_missing\_index\_details d

ON d.index\_handle = g.index\_handle

ORDER BY [Total Cost] DESC

It's important to carefully analyze all the information that is presented before you create the index. Sometimes, for example, it's better to change the query instead of creating a new index.

The most common problem with the information is the *‘* *included\_columns* *’* field. It is very common to find queries that request certain fields unnecessarily: these then have to be supported by indexes. Most of times the better solution is to change the query to include fewer fields than create an index with too many included columns.

**Auto-growth**

Auto-growth is a good feature that ensures that the DBA will never find out that the database stopped because the data or log files hasn't enough space. Without a doubt, it's a good practice to keep auto-growth enabled, but you, as a DBA, should retain control over when and why auto-growth happens.

As well as the regular and planned database growth that allows the DBA to calculate days in advance when the auto-growth will happen, there are certain events that can precipitate auto-growth through making large temporary or sudden demands for disk space. Batch processes, for example, can use a large amount of space in data and log files and thereby trigger the auto-growth.

If this happens in the data files, the sudden auto-growth event will cause performance problems for the execution of the batch. The batch that triggered the auto-growth will then suffer performance problems while the auto-growth is happening. Sometimes the person who created the batch hasn’t full knowledge about the database server, probably not even knowing that the batch could run considerably faster if the data file size was sufficient, which would need to be planned in advance.

On the other hand, if the auto-growth happens in the log file, the growth will create new VLFs inside the log. If the log files has too many VLFs then the log backups and database recovery will get slower.

To avoid this, the DBA should always check whether unplanned auto-growth is happening, and if so then identify the reason and adjust the plans for auto-growth.

How do you check on the pattern of file-growth? SQL Server has a default trace that captures information about the file growth. If the default trace is running - and it is by default - you can recover this information.

First you need to recover the path where the trace is storing its files. You can do that with a small query over **sys.traces**. Next, we need to use the DMF **sys.fn\_trace\_gettable** to read the file and return the information as a table.

The query to find out the auto-growth that happened in our server is the following:

DECLARE @path NVARCHAR(260);

SELECT

@path = REVERSE(SUBSTRING(REVERSE([path]),

CHARINDEX('\', REVERSE([path])), 260)) + N'log.trc'

FROM sys.traces

WHERE is\_default = 1;

SELECT

DatabaseName,

[FileName],

SPID,

Duration,

StartTime,

EndTime,

FileType = CASE EventClass

WHEN 92 THEN 'Data'

WHEN 93 THEN 'Log'

END

FROM sys.fn\_trace\_gettable(@path, DEFAULT)

WHERE

EventClass IN (92,93)

ORDER BY

StartTime DESC;

**Index Fragmentation**

It is generally a good idea to choose ever-increasing fields as clustered keys. If, for any reason, a database doesn't follow this pattern, index fragmentation will happen and you then need to check the fragmentation level of the indexes.

When a new record is inserted in the middle of existing records, a new page is allocated for the table and some records are moved from an existing page to the new page. This procedure is called page split and one of the consequences is that the table pages will not be sequentially organized inside the data file any more. In other words, the data pages of the table will be fragmented.

The DBA work round the problem by using configuration of the fill factor in the indexes that are subject to fragmentation. The fill factor determines the amount of the page that will be filled, leaving space for new records. The DBA has the task to manage the interval between the executions of the index maintenance in a way that index maintenance happens before the pages run out of free space for new records.

It's the DBA’s task to balance the interval between the execution of indexes maintenances and the fill factor value. It's important to check the fragmentation of the indexes to be sure that you achieved the correct balance between these values, and that fragmentation isn't happening.

We can check the fragmentation of the indexes using the DMF **sys.dm\_db\_index\_physical\_stats**. This DMF can check information of a specific object or about all objects in a database. The first four parameters are the database id, table id, index id and partition id. The Table, index and partition parameters are optional.

The fifth parameter specifies how detailed the analysis should be. The options are 'LIMITED', 'SAMPLED' and 'DETAILED'. The indexes are hierarchical, the fragmentation can happen in any level of the hierarchy. The most common and important fragmentation is in the leaf level and we can check the leaf level fragmentation using the 'LIMITED' option, which runs faster and with lower impact for the server.

We need to do a join with **sys.indexes** to get the name of the index and with sys.tables so we can use the field **is\_ms\_shipped** to exclude the system objects from the result. We also filter the result by the field **index\_type\_desc** looking only for clustered and nonclustered index, because this query would otherwise include heaps and xml indexes, and do some calculations in the predicate to exclude too small indexes, because these will always show some level of fragmentation.

The query to check the fragmentation in the leaf level of the indexes will be this:

SELECT object\_name(IPS.object\_id) AS [TableName],

SI.name AS [IndexName],

IPS.Index\_type\_desc,

IPS.avg\_fragmentation\_in\_percent,

IPS.fragment\_count,

IPS.avg\_fragment\_size\_in\_pages,

alloc\_unit\_type\_desc

FROM sys.dm\_db\_index\_physical\_stats(db\_id(N'AdventureWorks2012'), NULL, NULL, NULL , 'LIMITED') IPS

JOIN sys.tables ST WITH (nolock) ON IPS.object\_id = ST.object\_id

JOIN sys.indexes SI WITH (nolock) ON IPS.object\_id = SI.object\_id AND IPS.index\_id = SI.index\_id

WHERE ST.is\_ms\_shipped = 0 and index\_type\_desc in ('NONCLUSTERED INDEX','CLUSTERED INDEX')

and (fragment\_count \* avg\_fragment\_size\_in\_pages) > 20

ORDER BY avg\_fragmentation\_in\_percent desc

We need a different query to check the fragmentation in other levels of the index. We need to use 'Detailed' mode instead of 'Limited' and we can exclude the information about the leaf level, since we already got this information using the 'Limited' mode.

The query to check the 'Detailed' fragmentation information will be this:

SELECT object\_name(IPS.object\_id) AS [TableName],

SI.name AS [IndexName],

IPS.Index\_type\_desc,

IPS.avg\_fragmentation\_in\_percent,

IPS.avg\_page\_space\_used\_in\_percent,

IPS.record\_count,

IPS.ghost\_record\_count,

IPS.fragment\_count,

IPS.avg\_fragment\_size\_in\_pages,

alloc\_unit\_type\_desc,

index\_level

FROM sys.dm\_db\_index\_physical\_stats(db\_id(N'AdventureWorks2012'), NULL, NULL, NULL , 'DETAILED') IPS

JOIN sys.tables ST WITH (nolock) ON IPS.object\_id = ST.object\_id

JOIN sys.indexes SI WITH (nolock) ON IPS.object\_id = SI.object\_id AND IPS.index\_id = SI.index\_id

WHERE ST.is\_ms\_shipped = 0 and (fragment\_count \* avg\_fragment\_size\_in\_pages) > 20

and index\_type\_desc in ('NONCLUSTERED INDEX','CLUSTERED INDEX') and index\_level<>0

ORDER BY avg\_fragmentation\_in\_percent desc

The most important field in both queries is **avg\_fragmentation\_in\_percent**. You can solve the fragmentation of the leaf level by doing a rebuild or by reorganizing the index by setting a new **fillfactor**, but **fillfactor** works only with leaf levels: To extend the fill factor to non-leaf levels you need to use **padIndex**.

**Job Status**

We often have several jobs defined in our SQL Servers and we need an easy way to check if any job has failed. Of course, we can check one by one, but a query to check if any of the jobs has failed can be very useful.

We will use two system tables from the MSDB database to retrieve the information about the jobs.

* **sysjobs:** This table has the information about the jobs
* **sysjobhistory:** This table has the information about all the executions of the jobs, but for each execution there is a row for each step of the job
* **sysjobactivity:** This table has the information about the last execution of each job and the last step executed for each job. Remember that the last step executed may not be the last existing step in many cases, for example, if the job failed.

A good result to retrieve is the last execution of each job, with the status of every step of the job. This information is in **sysjobhistory** table, but there is a problem: There isn't a field to identify each execution. We can order the records by the **job\_id**, **run\_date** (desc) and **run\_time** (desc), but we still need to get the records about the last execution. It will be a different number of records for each job, according to the last step executed.

The first step is a simple query to retrieve the last executed step for each job from **sysjobactivity** table. The query will be like this:

select job\_id,last\_executed\_step\_id

from msdb.dbo.sysjobactivity

where last\_executed\_step\_id is not null

For each job we retrieved with the query above, we need to retrieve the records about the steps of the job from **sysjobhistory** table. We can do this using Cross Apply in the query and using the field **last\_executed\_step\_id** as a parameter for the Top expression in the query. The query to retrieve the details of the last execution of each job will be like this:

with qry as

(select job\_id,last\_executed\_step\_id

from msdb.dbo.sysjobactivity

where last\_executed\_step\_id is not null)

select

job\_name, run\_status,

run\_date, run\_time,

run\_duration, step\_name, message

from qry

cross apply

(select top (qry.last\_executed\_step\_id + 1)

sysjobs.name as job\_name,

sysjobhistory.run\_status,

run\_date, run\_time,

run\_duration, step\_name,

message, step\_id

FROM msdb.dbo.sysjobhistory

INNER JOIN msdb.dbo.sysjobs

ON msdb.dbo.sysjobhistory.job\_id = msdb.dbo.sysjobs.job\_id

where msdb.dbo.sysjobs.job\_id=qry.job\_id

order by run\_date desc,run\_time desc) t

order by job\_name,step\_id

Finally, we need to translate the fields *‘* *run\_date* *’*, *‘* *run\_time* *’* and *‘* *run\_status* *’* to a more readable format. Let's use some functions to format these fields:

**run\_date:**

convert(date,convert(varchar,run\_date)) run\_date

**run\_time:**

Isnull(Substring(CONVERT(VARCHAR, run\_time + 1000000), 2, 2) + ':' +

Substring(CONVERT(VARCHAR, run\_time + 1000000), 4, 2)

+ ':' +

Substring(CONVERT(VARCHAR, run\_time + 1000000), 6, 2), '') as run\_time

**run\_status:**

CASE sysjobhistory.run\_status

WHEN 0 THEN 'Failed'

WHEN 1 THEN 'Succeeded'

WHEN 2 THEN 'Retry'

WHEN 3 THEN 'Cancelled'

END

AS

run\_status

To make the use of this query easier, we can filter the result for only the failed executions and create a view for this query. The view will be like this:

Create View LastFailedJobs as

with qry as

(select job\_id,last\_executed\_step\_id

from msdb.dbo.sysjobactivity

where last\_executed\_step\_id is not null)

select

job\_name,

CASE run\_status

WHEN 0 THEN 'Failed'

WHEN 1 THEN 'Succeeded'

WHEN 2 THEN 'Retry'

WHEN 3 THEN 'Cancelled'

END

AS

run\_status,

convert(date,convert(varchar,run\_date)) run\_date,

Isnull(Substring(CONVERT(VARCHAR, run\_time + 1000000), 2, 2) + ':' +

Substring(CONVERT(VARCHAR, run\_time + 1000000), 4, 2)

+ ':' +

Substring(CONVERT(VARCHAR, run\_time + 1000000), 6, 2), '') as run\_time,

run\_duration, step\_name, message

from qry

cross apply

(select top (qry.last\_executed\_step\_id + 1)

sysjobs.name as job\_name,

sysjobhistory.run\_status,

run\_date, run\_time,

run\_duration, step\_name,

message, step\_id

FROM msdb.dbo.sysjobhistory

INNER JOIN msdb.dbo.sysjobs

ON msdb.dbo.sysjobhistory.job\_id = msdb.dbo.sysjobs.job\_id

where msdb.dbo.sysjobs.job\_id=qry.job\_id

order by run\_date desc,run\_time desc) t

where run\_status<>1

order by job\_name,step\_id

**Memory Use**

Does our SQL Server has enough memory or our server is under memory pressure?

We need to check the buffer cache to identify if we have a good amount of cache hits and there isn't memory pressure.

We can do this by checking performance counters. Three good performance counters to check are:

* **Page Life Expectancy:** It’s Lifetime of the pages in the cache. The recommended value is over 300 sec.
* **Free List Stalls/sec:** The number of requests that have to wait for a free page. If this value is high, your server is under memory pressure.
* **Page Reads/sec:** If this counter has a high value this will confirm the memory pressure already highlighted by two counters

The query to retrieve these values will be like this:

SELECT object\_name, counter\_name, cntr\_value

FROM sys.dm\_os\_performance\_counters

WHERE [object\_name] LIKE '%Buffer Manager%'

AND [counter\_name] in ('Page life expectancy','Free list stalls/sec',

'Page reads/sec')

If you identify in the result that your server is under pressure, you have two ways to solve the problem:

* **Increase the server memory:** You can add more memory or adjust the configuration 'max server memory' if possible.
* **Adjust the queries:** You can optimize the queries of your applications, reducing the number of pages to read in each query and reducing the memory pressure.

**Next Steps**

These queries are just the beginning. You will need to plan how to execute these queries and to use other queries that I didn’t include here.

You will need to decide if you will use native features to collect information, such as Data Collector and SQL Agent Jobs or you will use monitoring tools such as [Redgate SQL Monitor](http://www.red-gate.com/products/dba/sql-monitor/resources/articles/sql-server-performance-monitoring-with-sql-monitor?__hstc=81186632.5f59805d6dc5a90dd61565d83d824dbb.1467725282740.1467725282740.1468246626000.2&__hssc=81186632.1.1468246626000&__hsfp=1143474557). I wrote an article about Data Collector [here](https://www.simple-talk.com/sql/database-administration/centralize-your-database-monitoring-process/).

Here are some related resources on monitoring that I’ve written:

* [Are the statistics being updated?](https://www.simple-talk.com/blogs/2016/04/27/are-the-statistics-being-updated/)
* [How to identify contention in TempDb](https://www.simple-talk.com/blogs/2016/05/09/how-to-identify-contention-in-tempdb/)
* [How to find CPU-intensive queries](https://www.simple-talk.com/blogs/2016/04/19/7597/)
* [Exploring query plans in SQL](https://www.simple-talk.com/sql/database-administration/exploring-query-plans-in-sql/)
* [Checking the plan cache warnings for a SQL Server database](https://www.simple-talk.com/sql/t-sql-programming/checking-the-plan-cache-warnings-for-a-sql-server-database/)
* [Identifying and solving index scan problems](https://www.simple-talk.com/sql/performance/identifying-and-solving-index-scan-problems/)

Thank this author by sharing:

[Search for other articles by Dennes Torres](https://www.simple-talk.com/author/dennes-torres/)

Top of Form

|  |  |
| --- | --- |
| **Subject:** | **My 50cts** |
| **Posted by:** | *Wilfred van Dijk* (not signed in) |
| **Posted on:** | *Thursday, June 2, 2016 at 5:45 AM* |
| **Message:** | You can convert the date and time format in MSDB to a datetime format using the dbo.agent\_datetime() function  300 seconds is so 1995... I recommend this article: http://www.sqlskills.com/blogs/paul/page-life-expectancy-isnt-what-you-think/ |

|  |  |
| --- | --- |
| **Subject:** | **Index Fragmentation** |
| **Posted by:** | *Anonymous* (not signed in) |
| **Posted on:** | *Monday, June 13, 2016 at 12:45 AM* |
| **Message:** | dO U MEAN patindex OR PADINDEX? |

|  |  |
| --- | --- |
| **Subject:** | **Index Fragmentation** |
| **Posted by:** | *Anonymous* (not signed in) |
| **Posted on:** | *Monday, June 13, 2016 at 4:49 AM* |
| **Message:** | Hi,  I mean PAD\_Index   Cheers, |

|  |  |
| --- | --- |
| **Subject:** | **Sql2008-R2 - Baseline perf counters** |
| **Posted by:** | *medbiller* ([view profile](https://www.simple-talk.com/community/user/Profile.aspx?UserID=46816)) |
| **Posted on:** | *Tuesday, July 5, 2016 at 2:40 PM* |
| **Message:** | Grate work! Sql 2008-R2 does'n,t like sys.dm\_db\_index\_physical\_stats(db\_id(N'DATABASENAME'), ... so had to substitute db\_id(N'DATABASENAME') for the corresponding integer (i.e. 14).  I often read to do a benchmark on your servers before tinkering with them to be able and compare before & after scenarios. May I suggest you write an article (much like this one) outlining a basic, simple but comprehensive enough set of benchmark counters and how to go about implementing then?  Thanks. |

|  |  |
| --- | --- |
| **Subject:** | **How can I find query that caused an entry in 'Missing Index Rows'** |
| **Posted by:** | *medbiller* ([view profile](https://www.simple-talk.com/community/user/Profile.aspx?UserID=46816)) |
| **Posted on:** | *Tuesday, July 5, 2016 at 2:46 PM* |
| **Message:** | Before going straight to add the missing index it would be nice to be able to analyze the query causing the corresponding row in missing querys rows. Can this be done & how? |

|  |  |
| --- | --- |
| **Subject:** | **Re: Medbiller** |
| **Posted by:** | *Dennes Torres* ([view profile](https://www.simple-talk.com/community/user/Profile.aspx?UserID=215743)) |
| **Posted on:** | *Tuesday, July 5, 2016 at 9:41 PM* |
| **Message:** | Hi,  I think you will like to read this article: https://www.simple-talk.com/sql/database-administration/centralize-your-database-monitoring-process/  The missing index information is collected by the query optimizer. When the QO identifies a missing index it generates a plan warning and register the information.  You can find plan warnings analyzing the plan cache. This other article explains how: https://www.simple-talk.com/sql/t-sql-programming/checking-the-plan-cache-warnings-for-a-sql-server-database/  Cheers, |

|  |  |
| --- | --- |
| **Subject:** | **Baseline information** |
| **Posted by:** | *sterbalr* ([view profile](https://www.simple-talk.com/community/user/Profile.aspx?UserID=213351)) |
| **Posted on:** | *Friday, July 8, 2016 at 7:45 AM* |
| **Message:** | Glen Berry has posted some useful queries for baselining:  http://www.sqlskills.com/blogs/glenn/sql-server-diagnostic-information-queries-for-july-2016/ |

Bottom of Form