# Tuning Database Performance

## SQL Server: Monitoring Disk Usage

Microsoft SQL Server performance depends heavily on the I/O Subsystem (IOS). Latency in the IOS can result in many performance problems.

For example, you may experience slow response times and problems caused by tasks timing out.

It is critical that you monitor your disk usage.

Disk counters that you can monitor to determine disk activity are divided into the following two groups:

1.Primary

PhysicalDisk: Avg. Disk sec/Write

PhysicalDisk: Avg. Disk sec/Read

2.Secondary

PhysicalDisk: Avg. Disk Queue Length

PhysicalDisk: Disk Bytes/sec

PhysicalDisk: Disk Transfers/sec

Primary - PhysicalDisk

In System Monitor, the Avg. Disk sec/Write and Avg. Disk sec/Read counters are considered “primary”.

These counters should be examined first and do not need any additional information to evaluate the drive performance.

These counters determine the average latency of an I/O request.

Avg. Disk Sec/Read is the average time in seconds of a read of data from the disk.

The following list shows ranges of possible values and what the ranges represent:

1. Less than 10 ms - very good
2. 2.Between 10 - 20 ms - okay
3. 3.Between 20 - 50 ms - slow, needs attention
4. 4.Greater than 50 ms – Serious I/O bottleneck

Avg. Disk Sec/Write is the average time in seconds of a write of data to the disk. The guidelines for the Avg. Disk Sec/Read values apply here

Introduction

The server performance depends on various factors such as CPU, Memory, and Storage configurations.

SQL Server works on top of the operating system and works closely with these resources.

Usually, if you RDP to SQL Server, you would probably notice that SQL Server is a top memory consumer even on an idle instance.

It is a pretty common question for those unfamiliar with how SQL Server memory management works.

SQL Server memory management is dynamic, and DBA does not require a specific memory configuration for each memory component.

SQL Server uses buffer cache to load pages from the disk depending on the workload requirements.

It is necessary to minimize the disk I/O requirements. As per Microsoft document, SQL Server works on balancing the following goals:

Keep the buffer pool from becoming so big that the entire system is low on memory

Minimize physical I/O to the database files by maximizing the size of the buffer pool

By default, SQL Server dynamically acquires memory depending on the server RAM; however, it ensures that it does not create memory shortage for remaining processes.

SQL Server database engine has a background process called Resource monitor.

Its task is to monitor the internal and external memory indicators.

## Resource Governor

SQL Server Resource Governor is a feature that you can use to manage SQL Server workload and system resource consumption.

Resource Governor enables you to specify limits on the amount of CPU, physical I/O, and memory that incoming application requests can use.

\*\*Signal versus resource waits\*\*

SELECT

SUM(signal\_wait\_time\_ms) AS [Signal Wait Time (ms)]

,CONVERT(DECIMAL(7,4), 100.0 \* SUM (signal\_wait\_time\_ms) /

SUM(wait\_time\_ms)) AS [% Signal waits]

,SUM(wait\_time\_ms - signal\_wait\_time\_ms) AS [Resource Wait Time

(ms)]

,CONVERT (DECIMAL(7,4), 100.0 \* sum(wait\_time\_ms -

signal\_wait\_time\_ms) /

SUM(wait\_time\_ms)) AS [% Resource waits]

FROM

sys.dm\_os\_wait\_stats WITH (NOLOCK);

SQL Server latches are an internal SQL Server mechanism that serves to protect shared memory resources, like pages and memory data structures inside the buffer pool, in order to coordinate access to those resources and protect them from corruption.

There are several errors that may occur during I/O requests. All of them are

severe, and you need to set up alerts in the system for them.

1. Error 823 indicates that the OS I/O API call was not successful.

This is often a sign of hardware issues.

2. Errors 605 and 824 indicate logical consistency issues with the

data pages. When you encountered either of these errors,

immediately check whether the database is corrupted, using the

DBCC CHECKDB command. You may also encounter those errors

in case of faulty I/O drivers, which can corrupt data pages during

transfer.

3. Error 833 tells you that an I/O request (OS API call) took longer

than 15 seconds to return. This is abnormal; check the health of the

disk subsystem when you see this error.

4. Error 825 indicates that an I/O request failed and had to be retried

in order to succeed. As with Error 833, check the health of the disk

subsystem.

sys.dm\_io\_virtual\_file\_stats is a dynamic management view in SQL Server that provides information about the input/output (I/O) activity of database files.

This view can be used to monitor the performance of the storage subsystem and identify potential bottlenecks or issues.

Latches are used to ensure the consistency and integrity of data in the database and to prevent data corruption or other issues that can occur when multiple threads attempt to access the same data concurrently

A PAGEIOLATCH is a type of latch in SQL Server that is used to protect data pages as they are read from or written to disk.

A PAGEIOLATCH is typically acquired when a data page is brought into memory from disk, and it is held until the page is no longer needed or until it is written back to disk

SELECT r.session\_id, r.blocking\_session\_id, r.command,

r.wait\_type, r.wait\_time, r.wait\_resource, r.percent\_complete,

s.login\_name, s.cpu\_time, s.memory\_usage, s.program\_name

FROM sys.dm\_io\_pending\_io\_requests AS r

INNER JOIN sys.dm\_exec\_sessions AS s

ON r.session\_id = s.session\_id

WHERE r.wait\_type = 'PAGEIOLATCH\_SH'

using the sys.dm\_io\_pending\_io\_requests dynamic management view to identify queries that are waiting on PAGEIOLATCH latches.

# The sys.dm\_io\_virtual\_file\_stats view is the most important tool in SQL Server I/O performance troubleshooting.

CREATE TABLE #Snapshot

(

database\_id SMALLINT NOT NULL,

file\_id SMALLINT NOT NULL,

num\_of\_reads BIGINT NOT NULL,

num\_of\_bytes\_read BIGINT NOT NULL,

io\_stall\_read\_ms BIGINT NOT NULL,

num\_of\_writes BIGINT NOT NULL,

num\_of\_bytes\_written BIGINT NOT NULL,

io\_stall\_write\_ms BIGINT NOT NULL

);

GO

INSERT INTO

#Snapshot(database\_id,file\_id,num\_of\_reads,num\_of\_bytes\_read

,io\_stall\_read\_ms,num\_of\_writes,num\_of\_bytes\_written,io\_stall\_write\_ms)

SELECT database\_id,file\_id,num\_of\_reads,num\_of\_bytes\_read

,io\_stall\_read\_ms,num\_of\_writes,num\_of\_bytes\_written,io\_stall\_write\_ms

FROM sys.dm\_io\_virtual\_file\_stats(NULL,NULL)

OPTION (RECOMPILE);

-- Set test interval (1 minute). Use larger intervals in production

WAITFOR DELAY '00:01:00.000';

;WITH Stats(db\_id, file\_id, Reads, ReadBytes, Writes

,WrittenBytes, ReadStall, WriteStall)

as

(

SELECT

s.database\_id, s.file\_id ,fs.num\_of\_reads - s.num\_of\_reads

,fs.num\_of\_bytes\_read - s.num\_of\_bytes\_read

,fs.num\_of\_writes - s.num\_of\_writes

,fs.num\_of\_bytes\_written - s.num\_of\_bytes\_written

,fs.io\_stall\_read\_ms - s.io\_stall\_read\_ms

,fs.io\_stall\_write\_ms - s.io\_stall\_write\_ms

FROM

#Snapshot s JOIN

sys.dm\_io\_virtual\_file\_stats(NULL, NULL) fs ON

s.database\_id = fs.database\_id and

s.file\_id = fs.file\_id

)

SELECT

s.db\_id AS [DB ID], d.name AS [Database]

,mf.name AS [File Name], mf.physical\_name AS [File Path]

,mf.type\_desc AS [Type], s.Reads

,CONVERT(DECIMAL(12,3), s.ReadBytes / 1048576.) AS [Read

MB]

,CONVERT(DECIMAL(12,3), s.WrittenBytes / 1048576.) AS

[Written MB]

,s.Writes, s.Reads + s.Writes AS [IO Count]

,CONVERT(DECIMAL(5,2),100.0 \* s.ReadBytes /

(s.ReadBytes + s.WrittenBytes)) AS [Read %]

,CONVERT(DECIMAL(5,2),100.0 \* s.WrittenBytes /

(s.ReadBytes + s.WrittenBytes)) AS [Write

%]

,s.ReadStall AS [Read Stall]

,s.WriteStall AS [Write Stall]

,CASE WHEN s.Reads = 0

THEN 0.000

ELSE CONVERT(DECIMAL(12,3),1.0 \* s.ReadStall /

s.Reads)

END AS [Avg Read Stall]

,CASE WHEN s.Writes = 0

THEN 0.000

ELSE CONVERT(DECIMAL(12,3),1.0 \* s.WriteStall /

s.Writes)

END AS [Avg Write Stall]

FROM

Stats s JOIN sys.master\_files mf WITH (NOLOCK) ON

s.db\_id = mf.database\_id and

s.file\_id = mf.file\_id

JOIN sys.databases d WITH (NOLOCK) ON

s.db\_id = d.database\_id

WHERE -- Only display files with more than 20MB throughput

(s.ReadBytes + s.WrittenBytes) > 20 \* 1048576

ORDER BY

s.db\_id, s.file\_id

OPTION (RECOMPILE);

A scheduler is a component that determines which tasks should be executed and when they should be executed, based on the available resources and the priorities of the tasks.

Types of checkpoints:

1. Internal Checkpoint: Internal checkpoints are initiated automatically by the DBMS and

do not require any action by the database administrator (DBA).

2. Manual Checkpoint: These are created by the database administrator (DBA) using the CHECKPOINT or BACKUP LOG WITH NO\_LOG commands,

or through the SQL Server Management Studio interface. Manual checkpoints are used to force the database engine to create a checkpoint

at a specific point in time, which can be useful for tasks such as backing up the database or preparing it for a maintenance operation.

3. Automatic Checkpoint: These are created by the SQL Server database engine at regular intervals or when certain conditions are met,

such as when a certain amount of log data has been generated or when a certain amount of time has elapsed since the last checkpoint.

Automatic checkpoints are used to ensure that the transaction log does not grow too large, which can affect the performance of the database.

4. Indirect Checkpoint: From SQL Server 2012, we have another option: indirect checkpoint. With this method, SQL Server tries to balance I/O load by

executing checkpoints much more frequently – in some cases, even continuously. This helps to mitigate bursts of data writes,

making the I/O load much more balanced. Use it instead of automatic checkpoint whenever possible.

Enabling indirect checkpoint in the system immediately changed the I/O pattern, making it much more balanced.

PAGEIOLATCH is a type of wait that occurs when SQL Server is waiting for a page to be read from or written to disk. A page is the unit of data that is read from or written to disk in SQL Server. When a page is needed in memory, but is not currently in memory, SQL Server must wait for it to be read from disk into memory. This wait is called a PAGEIOLATCH wait.

PAGEIOLATCH waits can be a normal part of database operation, especially in heavily-used databases. However, excessive PAGEIOLATCH waits can indicate that the database is not performing optimally, and may be a sign of issues such as insufficient memory, insufficient I/O capacity, or poorly designed queries.

To troubleshoot and resolve PAGEIOLATCH waits, you can try the following strategies:

1. Monitor the PAGEIOLATCH wait statistics to identify the most heavily-waited pages and the queries that are causing the waits.

2. Check the current memory configuration and increase the amount of available memory if necessary.

3. Review and optimize the queries that are causing the PAGEIOLATCH waits.

4. Consider adding more I/O capacity to the system, such as by adding more disks or using faster disks.

5. Consider using features such as data compression and partitioning to reduce the number of pages that need to be read from or written to disk.

Technically, there are six such waits, but only three are typically

present in the system:

1. PAGEIOLATCH\_SH: This type of wait occurs when SQL Server is waiting for a shared (read) latch on a page.

This can occur when multiple queries need to read the same page at the same time.

2. PAGEIOLATCH\_EX: This type of wait occurs when SQL Server is waiting for an exclusive (write) latch on a page.

This can occur when a query needs to modify the page, or when multiple queries need to update different parts of the same page at the same time.

3. PAGEIOLATCH\_UP: This type of wait occurs when SQL Server is waiting for a latch on a page that is being updated.

This can occur when a query is updating a page and another query needs to read or modify the page at the same time.

### 3. Getting Page Life Expectancy in the system

SELECT object\_name, counter\_name, instance\_name, cntr\_value as [PLE(sec)]

FROM sys.dm\_os\_performance\_counters

WITH (NOLOCK) WHERE counter\_name = 'Page life expectancy'

OPTION (RECOMPILE);

Troubleshooting Checklist

Troubleshoot the following:

* Analyse disk subsystem latency with the sys.dm\_io\_virtual\_file\_stats view
* Check if high latency is caused by bursts in I/O activity by analyzing SQL Server and OS performance counters.
* Review I/O metrics at the VM and storage levels, paying attention to noisy neighbors in your setup.
* Check disk queue depth settings in the I/O stack.
* Troubleshoot SQL Server checkpoint performance and switch to indirect checkpoints.
* Troubleshoot log performance if you see significant WRITELOG waits (see Chapter 11).
* Troubleshoot tempdb performance if you see significant IO\_COMPLETION waits and high tempdb usage and latency (see Chapter 9).
* Detect and optimize inefficient queries if you see high PAGEIOLATCH waits in the system

## Detecting Inefficient Queries

When you troubleshoot a system, always analyse whether queries in the system are poorly optimized.

1. Plan cache-based execution statistics

Plan cache-based execution statistics are statistics that are gathered about the execution of queries in a database management system. These statistics are stored in the plan cache, which is a memory area in the database where the query execution plans are stored. The purpose of these statistics is to help the database optimizer make better decisions about how to execute queries.

Here are some examples of plan cache-based execution statistics:

1. Number of executions: This statistic tracks the number of times a query has been executed. For example, if a query has been executed 100 times, this statistic would be 100.
2. Execution time: This statistic tracks the total amount of time that has been spent executing a query. For example, if a query took a total of 10 seconds to execute over the course of 100 executions, this statistic would be 1000 seconds.
3. Average execution time: This statistic is calculated by dividing the execution time by the number of executions. In the example above, the average execution time would be 10 seconds.
4. CPU time: This statistic tracks the total amount of CPU time that has been spent executing a query.
5. Average CPU time: This statistic is calculated by dividing the CPU time by the number of executions.
6. Logical reads: This statistic tracks the number of data pages that have been read from disk while executing a query.
7. Physical reads: This statistic tracks the number of data pages that have been read from disk and placed into the database cache while executing a query.

Analyzing plan cache-based execution statistics is a limited detection technique. Nonetheless, it is very simple to use and, in many cases, adequate. It works in all SQL Server versions and is always present in the system. To collect the data, no additional monitoring is required.

You can get execution statistics using the sys.dm\_exec\_query\_stats view.

### Example. Using the sys.dm\_exec\_query\_stats view

SELECT TOP 50

qs.creation\_time AS [Cached Time]

,qs.last\_execution\_time AS [Last Exec Time]

,SUBSTRING(qt.text, (qs.statement\_start\_offset/2)+1,

((

CASE qs.statement\_end\_offset

WHEN -1 THEN DATALENGTH(qt.text)

ELSE qs.statement\_end\_offset

END - qs.statement\_start\_offset)/2)+1) AS SQL

,qp.query\_plan AS [Query Plan]

,qs.execution\_count AS [Exec Cnt]

,CONVERT(DECIMAL(10,5),

IIF(datediff(second,qs.creation\_time,

qs.last\_execution\_time) = 0,

NULL,

1.0 \* qs.execution\_count /

datediff(second,qs.creation\_time,

qs.last\_execution\_time)

)

) AS [Exec Per Second]

,(qs.total\_logical\_reads + qs.total\_logical\_writes) /

qs.execution\_count AS [Avg IO]

,(qs.total\_worker\_time / qs.execution\_count / 1000)

AS [Avg CPU(ms)]

,qs.total\_logical\_reads AS [Total Reads]

,qs.last\_logical\_reads AS [Last Reads]

,qs.total\_logical\_writes AS [Total Writes]

,qs.last\_logical\_writes AS [Last Writes]

,qs.total\_worker\_time / 1000 AS [Total Worker Time]

,qs.last\_worker\_time / 1000 AS [Last Worker Time]

,qs.total\_elapsed\_time / 1000 AS [Total Elapsed Time]

,qs.last\_elapsed\_time / 1000 AS [Last Elapsed Time]

,qs.total\_rows AS [Total Rows]

,qs.last\_rows AS [Last Rows]

,qs.total\_rows / qs.execution\_count AS [Avg Rows]

,qs.total\_physical\_reads AS [Total Physical Reads]

,qs.last\_physical\_reads AS [Last Physical Reads]

,qs.total\_physical\_reads / qs.execution\_count

AS [Avg Physical Reads]

,qs.total\_grant\_kb AS [Total Grant KB]

,qs.last\_grant\_kb AS [Last Grant KB]

,(qs.total\_grant\_kb / qs.execution\_count)

AS [Avg Grant KB]

,qs.total\_used\_grant\_kb AS [Total Used Grant KB]

,qs.last\_used\_grant\_kb AS [Last Used Grant KB]

,(qs.total\_used\_grant\_kb / qs.execution\_count)

AS [Avg Used Grant KB]

,qs.total\_ideal\_grant\_kb AS [Total Ideal Grant KB]

,qs.last\_ideal\_grant\_kb AS [Last Ideal Grant KB]

,(qs.total\_ideal\_grant\_kb / qs.execution\_count)

AS [Avg Ideal Grant KB]

,qs.total\_columnstore\_segment\_reads

AS [Total CSI Segments Read]

,qs.last\_columnstore\_segment\_reads

AS [Last CSI Segments Read]

,(qs.total\_columnstore\_segment\_reads / qs.execution\_count)

AS [AVG CSI Segments Read]

,qs.max\_dop AS [Max DOP]

,qs.total\_spills AS [Total Spills]

,qs.last\_spills AS [Last Spills]

,(qs.total\_spills / qs.execution\_count) AS [Avg Spills]

FROM

sys.dm\_exec\_query\_stats qs WITH (NOLOCK)

CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) qt

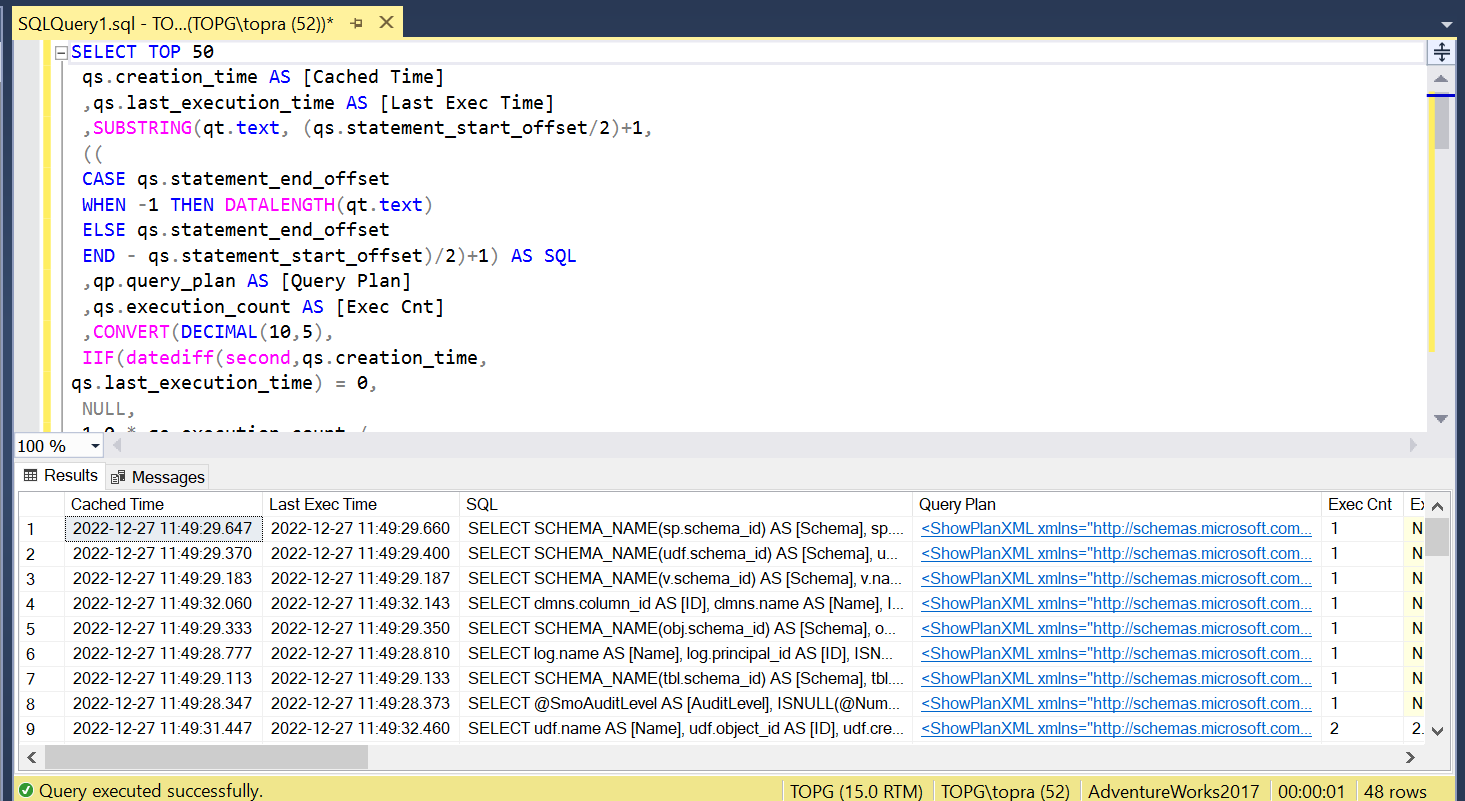
CROSS APPLY sys.dm\_exec\_query\_plan(qs.plan\_handle) qp

ORDER BY

[Avg IO] DESC

OPTION (RECOMPILE, MAXDOP 1);

Output:



Here is an example of how you could use the sys.dm\_exec\_query\_stats view to find the top 10 most expensive queries in terms of CPU time:

SELECT TOP 10

qs.total\_worker\_time AS [Total CPU Time],

qs.execution\_count AS [Execution Count],

qs.total\_elapsed\_time/qs.execution\_count AS [Average Elapsed Time],

qs.total\_worker\_time/qs.execution\_count AS [Average CPU Time],

qs.total\_logical\_reads/qs.execution\_count AS [Average Logical Reads],

qs.total\_physical\_reads/qs.execution\_count AS [Average Physical Reads],

qs.total\_elapsed\_time AS [Total Elapsed Time],

qs.total\_logical\_reads AS [Total Logical Reads],

qs.total\_physical\_reads AS [Total Physical Reads],

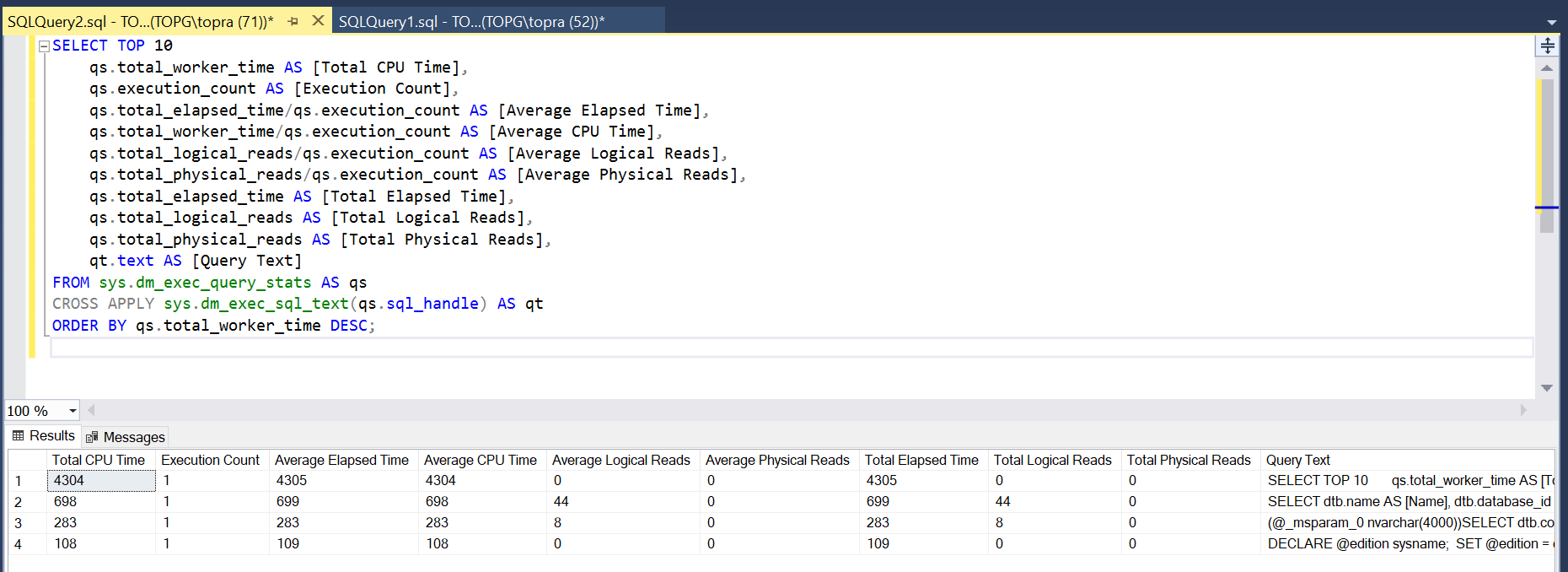
qt.text AS [Query Text]

FROM sys.dm\_exec\_query\_stats AS qs

CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) AS qt

ORDER BY qs.total\_worker\_time DESC;

Output:



1. Query\_hash and query\_plan\_hash in action

SELECT COUNT(\*) FROM sys.objects

GO

SELECT

qs.query\_hash, qs.query\_plan\_hash, qs.sql\_handle,

qs.plan\_handle,

SUBSTRING(qt.text, (qs.statement\_start\_offset/2)+1,

((

CASE qs.statement\_end\_offset

WHEN -1 THEN DATALENGTH(qt.text)

ELSE qs.statement\_end\_offset

END - qs.statement\_start\_offset)/2)+1

) as SQL

FROM

sys.dm\_exec\_query\_stats qs

CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) qt

ORDER BY query\_hash

OPTION (MAXDOP 1, RECOMPILE);

Output:



1. Example Using the sys.dm\_exec\_query\_stats view with query\_hash aggregation

;WITH Data

AS

(

SELECT TOP 50

qs.query\_hash

,COUNT(\*) as [Plan Count]

,MIN(qs.creation\_time) AS [Cached Time]

,MAX(qs.last\_execution\_time) AS [Last Exec Time]

,SUM(qs.execution\_count) AS [Exec Cnt]

,SUM(qs.total\_logical\_reads) AS [Total Reads]

,SUM(qs.total\_logical\_writes) AS [Total Writes]

,SUM(qs.total\_worker\_time / 1000) AS [Total Worker Time]

,SUM(qs.total\_elapsed\_time / 1000) AS [Total Elapsed Time]

,SUM(qs.total\_rows) AS [Total Rows]

,SUM(qs.total\_physical\_reads) AS [Total Physical Reads]

,SUM(qs.total\_grant\_kb) AS [Total Grant KB]

,SUM(qs.total\_used\_grant\_kb) AS [Total Used Grant KB]

,SUM(qs.total\_ideal\_grant\_kb) AS [Total Ideal Grant KB]

,SUM(qs.total\_columnstore\_segment\_reads)

AS [Total CSI Segments Read]

,MAX(qs.max\_dop) AS [Max DOP]

,SUM(qs.total\_spills) AS [Total Spills]

FROM

sys.dm\_exec\_query\_stats qs WITH (NOLOCK)

GROUP BY

qs.query\_hash

ORDER BY

SUM((qs.total\_logical\_reads + qs.total\_logical\_writes) /

qs.execution\_count) DESC

)

SELECT

d.[Cached Time]

,d.[Last Exec Time]

,d.[Plan Count]

,sql\_plan.SQL

,sql\_plan.[Query Plan]

,d.[Exec Cnt]

,CONVERT(DECIMAL(10,5),

IIF(datediff(second,d.[Cached Time], d.[Last Exec Time]) =

0,

NULL,

1.0 \* d.[Exec Cnt] /

datediff(second,d.[Cached Time], d.[Last Exec Time])

)

) AS [Exec Per Second]

,(d.[Total Reads] + d.[Total Writes]) / d.[Exec Cnt] AS [Avg IO]

,(d.[Total Worker Time] / d.[Exec Cnt] / 1000) AS [Avg CPU(ms)]

,d.[Total Reads]

,d.[Total Writes]

,d.[Total Worker Time]

,d.[Total Elapsed Time]

,d.[Total Rows]

,d.[Total Rows] / d.[Exec Cnt] AS [Avg Rows]

,d.[Total Physical Reads]

,d.[Total Physical Reads] / d.[Exec Cnt] AS [Avg Physical Reads]

,d.[Total Grant KB]

,d.[Total Grant KB] / d.[Exec Cnt] AS [Avg Grant KB]

,d.[Total Used Grant KB]

,d.[Total Used Grant KB] / d.[Exec Cnt] AS [Avg Used Grant KB]

,d.[Total Ideal Grant KB]

,d.[Total Ideal Grant KB] / d.[Exec Cnt] AS [Avg Ideal Grant KB]

,d.[Total CSI Segments Read]

,d.[Total CSI Segments Read] / d.[Exec Cnt] AS [AVG CSI Segments

Read]

,d.[Max DOP]

,d.[Total Spills]

,d.[Total Spills] / d.[Exec Cnt] AS [Avg Spills]

FROM

Data d

CROSS APPLY

(

SELECT TOP 1

SUBSTRING(qt.text, (qs.statement\_start\_offset/2)+1,

((

CASE qs.statement\_end\_offset

WHEN -1 THEN DATALENGTH(qt.text)

ELSE qs.statement\_end\_offset

END - qs.statement\_start\_offset)/2)+1

) AS SQL

,qp.query\_plan AS [Query Plan]

FROM

sys.dm\_exec\_query\_stats qs

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

qt

CROSS APPLY

sys.dm\_exec\_query\_plan(qs.plan\_handle) qp

WHERE

qs.query\_hash = d.query\_hash AND ISNULL(qt.text,'')

<> ''

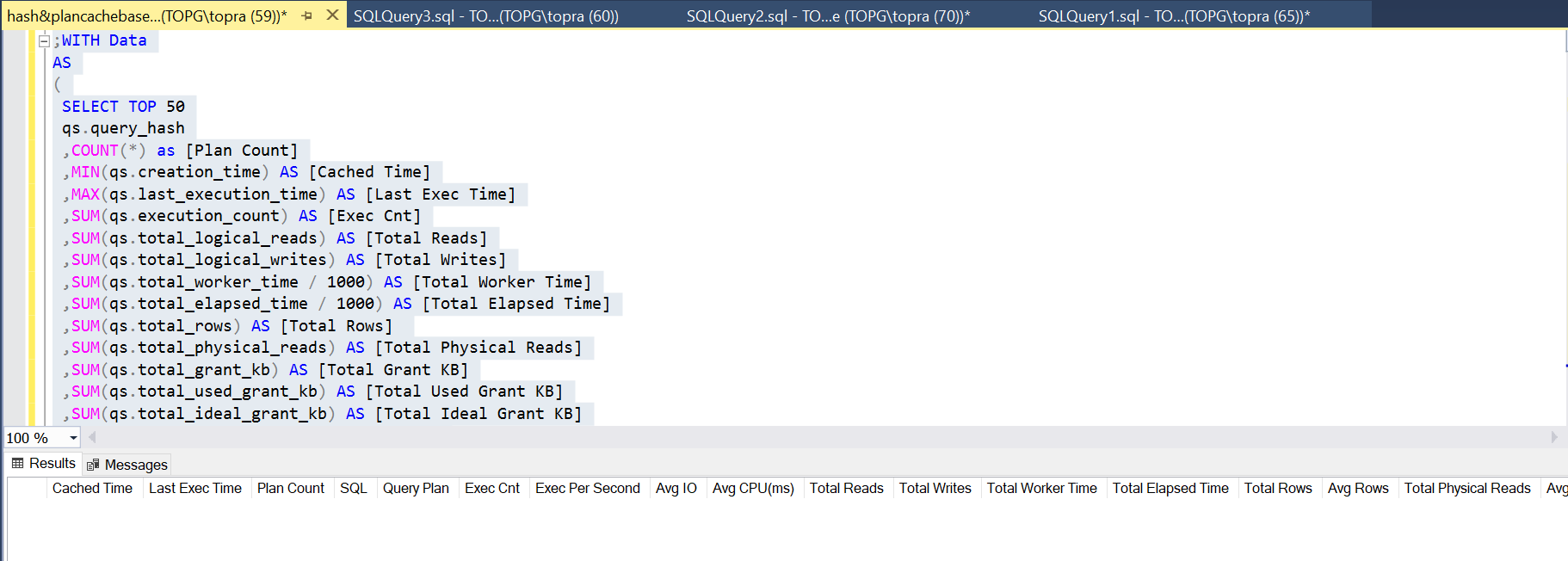
) sql\_plan

ORDER BY

[Avg IO] DESC

OPTION (RECOMPILE, MAXDOP 1);

Ouput:



## Using the sys.dm\_exec\_procedure\_stats view

Queries:

SELECT TOP 50

DB\_NAME(ps.database\_id) AS [DB]

,OBJECT\_NAME(ps.object\_id, ps.database\_id) AS [Proc Name]

,ps.type\_desc AS [Type]

,ps.cached\_time AS [Cached Time]

,ps.last\_execution\_time AS [Last Exec Time]

,qp.query\_plan AS [Plan]

,ps.execution\_count AS [Exec Count]

,CONVERT(DECIMAL(10,5),

IIF(datediff(second,ps.cached\_time, ps.last\_execution\_time)

= 0,

NULL,

1.0 \* ps.execution\_count /

datediff(second,ps.cached\_time,

ps.last\_execution\_time)

)

) AS [Exec Per Second]

,(ps.total\_logical\_reads + ps.total\_logical\_writes) /

ps.execution\_count AS [Avg IO]

,(ps.total\_worker\_time / ps.execution\_count / 1000)

AS [Avg CPU(ms)]

,ps.total\_logical\_reads AS [Total Reads]

,ps.last\_logical\_reads AS [Last Reads]

,ps.total\_logical\_writes AS [Total Writes]

,ps.last\_logical\_writes AS [Last Writes]

,ps.total\_worker\_time / 1000 AS [Total Worker Time]

,ps.last\_worker\_time / 1000 AS [Last Worker Time]

,ps.total\_elapsed\_time / 1000 AS [Total Elapsed Time]

,ps.last\_elapsed\_time / 1000 AS [Last Elapsed Time]

,ps.total\_physical\_reads AS [Total Physical Reads]

,ps.last\_physical\_reads AS [Last Physical Reads]

,ps.total\_physical\_reads / ps.execution\_count AS [Avg Physical

Reads]

,ps.total\_spills AS [Total Spills]

,ps.last\_spills AS [Last Spills]

,(ps.total\_spills / ps.execution\_count) AS [Avg Spills]

FROM

sys.dm\_exec\_procedure\_stats ps WITH (NOLOCK)

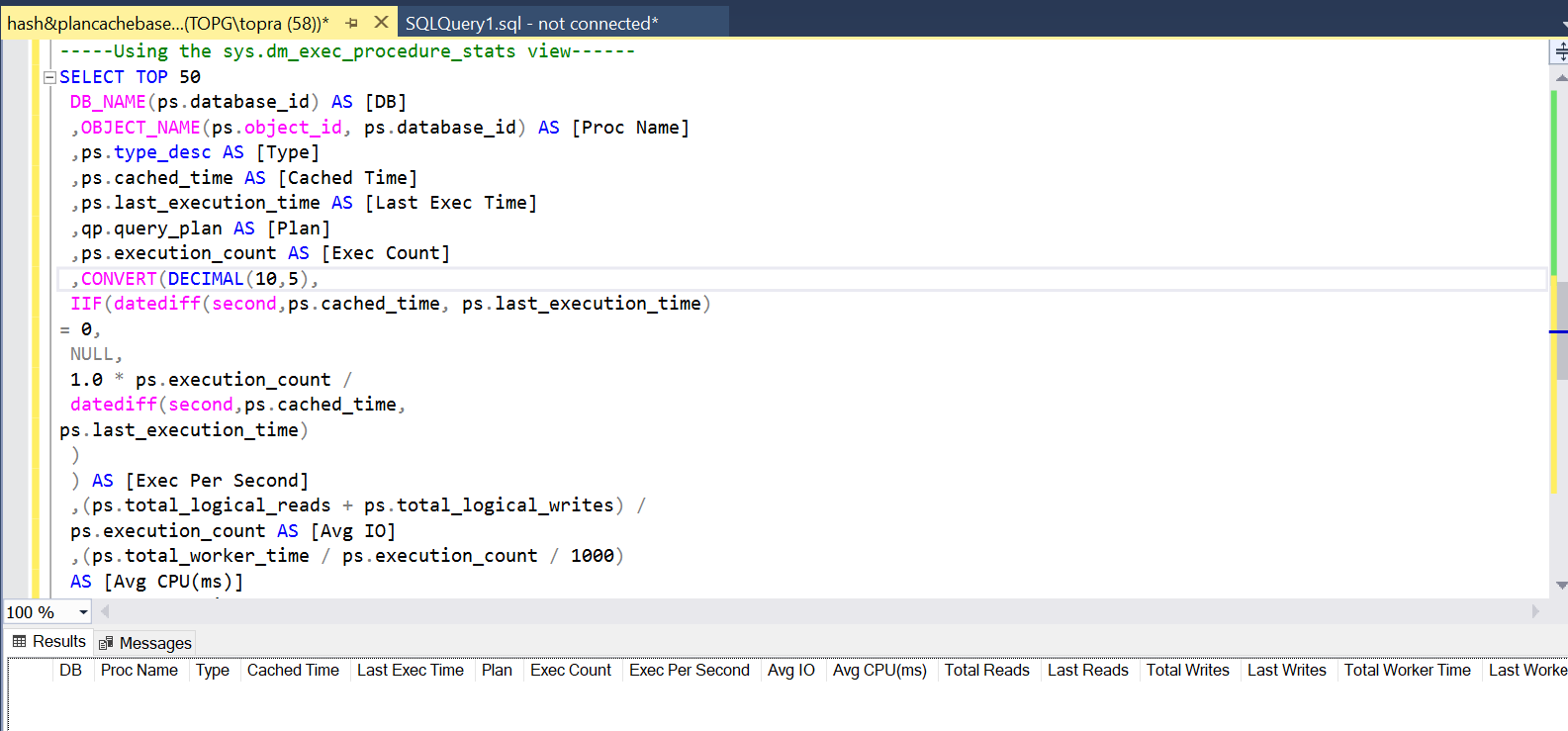
CROSS APPLY sys.dm\_exec\_query\_plan(ps.plan\_handle) qp

ORDER BY

[Avg IO] DESC

OPTION (RECOMPILE, MAXDOP 1);

Output:



1. Getting execution plan and statistics for stored procedure statements

SELECT

qs.creation\_time AS [Cached Time]

,qs.last\_execution\_time AS [Last Exec Time]

,SUBSTRING(qt.text, (qs.statement\_start\_offset/2)+1,

((

CASE qs.statement\_end\_offset

WHEN -1 THEN DATALENGTH(qt.text)

ELSE qs.statement\_end\_offset

END - qs.statement\_start\_offset)/2)+1) AS SQL

,qp.query\_plan AS [Query Plan]

,CONVERT(DECIMAL(10,5),

IIF(datediff(second,qs.creation\_time,

qs.last\_execution\_time) = 0,

NULL,

1.0 \* qs.execution\_count /

datediff(second,qs.creation\_time,

qs.last\_execution\_time)

)

) AS [Exec Per Second]

,(qs.total\_logical\_reads + qs.total\_logical\_writes) /

qs.execution\_count AS [Avg IO]

,(qs.total\_worker\_time / qs.execution\_count / 1000)

AS [Avg CPU(ms)]

,qs.total\_logical\_reads AS [Total Reads]

,qs.last\_logical\_reads AS [Last Reads]

,qs.total\_logical\_writes AS [Total Writes]

,qs.last\_logical\_writes AS [Last Writes]

,qs.total\_worker\_time / 1000 AS [Total Worker Time]

,qs.last\_worker\_time / 1000 AS [Last Worker Time]

,qs.total\_elapsed\_time / 1000 AS [Total Elapsed Time]

,qs.last\_elapsed\_time / 1000 AS [Last Elapsed Time]

,qs.total\_rows AS [Total Rows]

,qs.last\_rows AS [Last Rows]

,qs.total\_rows / qs.execution\_count AS [Avg Rows]

,qs.total\_physical\_reads AS [Total Physical Reads]

,qs.last\_physical\_reads AS [Last Physical Reads]

,qs.total\_physical\_reads / qs.execution\_count

AS [Avg Physical Reads]

,qs.total\_grant\_kb AS [Total Grant KB]

,qs.last\_grant\_kb AS [Last Grant KB]

,(qs.total\_grant\_kb / qs.execution\_count)

AS [Avg Grant KB]

,qs.total\_used\_grant\_kb AS [Total Used Grant KB]

,qs.last\_used\_grant\_kb AS [Last Used Grant KB]

,(qs.total\_used\_grant\_kb / qs.execution\_count)

AS [Avg Used Grant KB]

,qs.total\_ideal\_grant\_kb AS [Total Ideal Grant KB]

,qs.last\_ideal\_grant\_kb AS [Last Ideal Grant KB]

,(qs.total\_ideal\_grant\_kb / qs.execution\_count)

AS [Avg Ideal Grant KB]

,qs.total\_columnstore\_segment\_reads

AS [Total CSI Segments Read]

,qs.last\_columnstore\_segment\_reads

AS [Last CSI Segments Read]

,(qs.total\_columnstore\_segment\_reads / qs.execution\_count)

AS [AVG CSI Segments Read]

,qs.max\_dop AS [Max DOP]

,qs.total\_spills AS [Total Spills]

,qs.last\_spills AS [Last Spills]

,(qs.total\_spills / qs.execution\_count) AS [Avg Spills]

FROM

sys.dm\_exec\_query\_stats qs WITH (NOLOCK)

CROSS APPLY sys.dm\_exec\_sql\_text(qs.sql\_handle) qt

CROSS APPLY sys.dm\_exec\_text\_query\_plan

(qs.plan\_handle,qs.statement\_start\_offset,qs.statement\_end\_offset)

qp

WHERE

OBJECT\_NAME(qt.objectid, qt.dbid) = <SP Name>

ORDER BY qs.statement\_start\_offset, qs.statement\_end\_offset

OPTION (RECOMPILE, MAXDOP 1);

1. Getting execution statistics for user-defined functions and triggers

Queries:

SELECT TOP 50

DB\_NAME(fs.database\_id) AS [DB]

,OBJECT\_NAME(fs.object\_id, fs.database\_id) AS [Function]

,fs.type\_desc AS [Type]

,fs.cached\_time AS [Cached Time]

,fs.last\_execution\_time AS [Last Exec Time]

,qp.query\_plan AS [Plan]

,fs.execution\_count AS [Exec Count]

,CONVERT(DECIMAL(10,5),

IIF(datediff(second,fs.cached\_time, fs.last\_execution\_time)

= 0,

NULL,

1.0 \* fs.execution\_count /

datediff(second,fs.cached\_time,

fs.last\_execution\_time)

)

) AS [Exec Per Second]

,(fs.total\_logical\_reads + fs.total\_logical\_writes) /

fs.execution\_count AS [Avg IO]

,(fs.total\_worker\_time / fs.execution\_count / 1000) AS [Avg

CPU(ms)]

,fs.total\_logical\_reads AS [Total Reads]

,fs.last\_logical\_reads AS [Last Reads]

,fs.total\_logical\_writes AS [Total Writes]

,fs.last\_logical\_writes AS [Last Writes]

,fs.total\_worker\_time / 1000 AS [Total Worker Time]

,fs.last\_worker\_time / 1000 AS [Last Worker Time]

,fs.total\_elapsed\_time / 1000 AS [Total Elapsed Time]

,fs.last\_elapsed\_time / 1000 AS [Last Elapsed Time]

,fs.total\_physical\_reads AS [Total Physical Reads]

,fs.last\_physical\_reads AS [Last Physical Reads]

,fs.total\_physical\_reads / fs.execution\_count AS [Avg Physical

Reads]

FROM

sys.dm\_exec\_function\_stats fs WITH (NOLOCK)

CROSS APPLY sys.dm\_exec\_query\_plan(fs.plan\_handle) qp

ORDER BY

[Avg IO] DESC

OPTION (RECOMPILE, MAXDOP 1);

SELECT TOP 50

DB\_NAME(ts.database\_id) AS [DB]

,OBJECT\_NAME(ts.object\_id, ts.database\_id) AS [Function]

,ts.type\_desc AS [Type]

,ts.cached\_time AS [Cached Time]

,ts.last\_execution\_time AS [Last Exec Time]

,qp.query\_plan AS [Plan]

,ts.execution\_count AS [Exec Count]

,CONVERT(DECIMAL(10,5),

IIF(datediff(second,ts.cached\_time, ts.last\_execution\_time)

= 0,

NULL,

1.0 \* ts.execution\_count /

datediff(second,ts.cached\_time,

ts.last\_execution\_time)

)

) AS [Exec Per Second]

,(ts.total\_logical\_reads + ts.total\_logical\_writes) /

ts.execution\_count AS [Avg IO]

,(ts.total\_worker\_time / ts.execution\_count / 1000) AS [Avg

CPU(ms)]

,ts.total\_logical\_reads AS [Total Reads]

,ts.last\_logical\_reads AS [Last Reads]

,ts.total\_logical\_writes AS [Total Writes]

,ts.last\_logical\_writes AS [Last Writes]

,ts.total\_worker\_time / 1000 AS [Total Worker Time]

,ts.last\_worker\_time / 1000 AS [Last Worker Time]

,ts.total\_elapsed\_time / 1000 AS [Total Elapsed Time]

,ts.last\_elapsed\_time / 1000 AS [Last Elapsed Time]

,ts.total\_physical\_reads AS [Total Physical Reads]

,ts.last\_physical\_reads AS [Last Physical Reads]

,ts.total\_physical\_reads / ts.execution\_count AS [Avg Physical

Reads]

FROM

sys.dm\_exec\_trigger\_stats ts WITH (NOLOCK)

CROSS APPLY sys.dm\_exec\_query\_plan(ts.plan\_handle) qp

ORDER BY

[Avg IO] DESC

OPTION (RECOMPILE, MAXDOP 1);

## SQL Traces and Extended Events

/\*\*Listing 4-7 shows code to capture queries that consume more than 3,000ms

of CPU time or produce more than 10,000 logical reads or writes. This code

will work in SQL Server 2012 and above; it may require small

modifications in SQL Server 2008 due to the different way it works with the

file target.

Example 4-7. Capturing CPU- and I/O intensive queries\*\*/

CREATE EVENT SESSION [Expensive Queries]

ON SERVER

ADD EVENT

sqlserver.sql\_statement\_completed

(

ACTION

(

sqlserver.client\_app\_name

,sqlserver.client\_hostname

,sqlserver.database\_id

,sqlserver.plan\_handle

,sqlserver.sql\_text

,sqlserver.username

)

WHERE

(

(

cpu\_time >= 3000000 or -- Time in microseconds

logical\_reads >= 10000 or

writes >= 10000

) AND

sqlserver.is\_system = 0

)

),

ADD EVENT

sqlserver.rpc\_completed

(

ACTION

(

sqlserver.client\_app\_name

,sqlserver.client\_hostname

,sqlserver.database\_id

,sqlserver.plan\_handle

,sqlserver.sql\_text

,sqlserver.username

)

WHERE

(

(

cpu\_time >= 3000000 or

logical\_reads >= 10000 or

writes >= 10000

) AND

sqlserver.is\_system = 0

)

)

ADD TARGET

package0.event\_file

(

SET FILENAME = 'c:\ExtEvents\Expensive Queries.xel'

)

WITH

(

event\_retention\_mode=allow\_single\_event\_loss

,max\_dispatch\_latency=30 seconds

);

/\*\* You can parse the captured results with the code from Listing 4-8.

Example 4-8. Parsing collected xEvent data\*\*/

;WITH TargetData(Data, File\_Name, File\_Offset)

AS

(

SELECT CONVERT(xml,event\_data) AS Data, file\_name, file\_offset

FROM

sys.fn\_xe\_file\_target\_read\_file

('c:\extevents\Expensive Queries\*.xel',NULL,NULL,NULL)

)

,EventInfo([Event],[Event Time],[DB],[Statement],[SQL],[User Name]

,[Client],[App],[CPU Time],[Duration],[Logical Reads]

,[Physical Reads],[Writes],[Rows],[PlanHandle]

,File\_Name,File\_Offset)

as

(

SELECT

Data.value('/event[1]/@name','sysname') AS [Event]

,Data.value('/event[1]/@timestamp','datetime') AS [Event Time]

,Data.value('((/event[1]/data[@name="database\_id"]/value/text())

[1])','INT')

AS [DB]

,Data.value('((/event[1]/data[@name="statement"]/value/text())

[1])'

,'nvarchar(max)') AS [Statement]

,Data.value('((/event[1]/data[@name="sql\_text"]/value/text())

[1])'

,'nvarchar(max)') AS [SQL]

,Data.value('((/event[1]/data[@name="username"]/value/text())

[1])'

,'nvarchar(255)') AS [User Name]

,Data.value('((/event[1]/data[@name="client\_hostname"]/value/text())

[1])'

,'nvarchar(255)') AS [Client]

,Data.value('((/event[1]/data[@name="client\_app\_name"]/value/text())

[1])'

,'nvarchar(255)') AS [App]

,Data.value('((/event[1]/data[@name="cpu\_time"]/value/text())

[1])'

,'bigint') AS [CPU Time]

,Data.value('((/event[1]/data[@name="duration"]/value/text())

[1])'

,'bigint') AS [Duration]

,Data.value('((/event[1]/data[@name="logical\_reads"]/value/text())

[1])'

,'int') AS [Logical Reads]

,Data.value('((/event[1]/data[@name="physical\_reads"]/value/text())

[1])'

,'int') AS [Physical Reads]

,Data.value('((/event[1]/data[@name="writes"]/value/text())[1])'

,'int') AS [Writes]

,Data.value('((/event[1]/data[@name="row\_count"]/value/text())

[1])'

,'int') AS [Rows]

,Data.value(

'xs:hexBinary(((/event[1]/action[@name="plan\_handle"]/value/text())

[1]))'

,'varbinary(64)') AS [PlanHandle]

,File\_Name

,File\_Offset

FROM

TargetData

)

SELECT

ei.\*, qp.Query\_Plan

FROM

EventInfo ei

OUTER APPLY sys.dm\_exec\_query\_plan(ei.PlanHandle) qp

OPTION (MAXDOP 1, RECOMPILE);

So far in this chapter, we have discussed two approaches to detecting inefficient queries. Both have limitations. Plan-cache-based data may miss some queries; SQL Traces and xEvents require you to perform complex analysis of the output and may have significant performance overhead in busy systems.

## Query Store

Query Store were first introduced in SQL Server 2016, to address the limitations (of plan-cache-based & SQL Traces & xEvents). When the Query store is enabled, SQL server captures and persists runtime statistics and execution plans of the queries in the database. It shows how the execution plans perform and how they evolve over time. Finally, it enables you to apply specific execution plans to queries that deal with parameter sniffing issues.

**NOTE**: The Query Store is disabled by default in the on-premises version of SQL Server. It is enabled by default in Azure SQL Databases and Azure SQL Managed Instances.

Query Store still adds overhead to the system. Just how much overhead depends on two main factors: the number of compilations and the data collection settings.

The more compilations SQL Server performs, the more load the Query Store must handle. In particular, the Query Store may not work very well in systems that have a very heavy, ad-hoc, non-parameterized workload.

Query Store’s configurations allow you to specify if you want to capture all queries or just expensive ones, along with aggregation intervals and data retention settings. If you collect more data and/or use smaller aggregation intervals, you’ll have more overhead.

The Query Store typically introduces only minor overhead. However, in some cases, it may be significant. I've been using the Query Store, for example, to troubleshoot the performance of one process that consists of a large number of small ad-hoc queries. I used QUERY CAPTURE MODE=ALL to capture all queries in the system, collecting nearly 10GB of data in the Query Store. The process took 8 hours with the Query Store enabled, versus 2.5 hours without it.

Nonetheless, if your system can handle the overhead, I recommend enabling Query Store. Some SQL Server features, such as Intelligent Query Processing, rely on and benefit from Query Store data.

We can interact with Query Store in two ways: through SSMS Graphics UI or directly querying data management views.

Let’s look at the UI first.

1. Query Store SSMS Reports

You’ll see a ***Query Store*** Folder in the ***Object Explorer.***

It contains a seven reports as mentioned name:

1. Regressed Queries
2. Overall Resource Consumption
3. Top Resource Consuming Queries
4. Queries With Forced Plans
5. Queries With High Variation
6. Query Wait Statistics
7. Tracked Queries
8. Working with Query Store DMVs

Here is a list of some of the DMVs and functions that you can use to work with Query Store in SQL Server:

1. sys.query\_store\_query: This DMV provides information about the queries captured in Query Store.
2. sys.query\_store\_plan: This DMV provides information about the execution plans captured in Query Store.
3. sys.query\_store\_runtime\_stats /sys.query\_store\_runtime\_stats\_interval: This DMV provides runtime statistics for queries captured in Query Store.
4. sys.query\_store\_wait\_stats: This DMV provides wait statistics for queries captured in Query Store.
5. sys.query\_context\_settings: This DMV provides context settings for queries captured in Query Store.
6. sys.query\_store\_query\_text: This DMV provides the text of the queries captured in Query Store.
7. sys.query\_store\_force\_plan: This function allows you to force the use of a specific execution plan for a query in Query Store.

/\*\*\*Example 4-9. Getting information about expensive queries from Query Store\*\*\*/

SELECT TOP 10

q.query\_id, qt.query\_sql\_text, qp.plan\_id, qp.query\_plan

,SUM(rs.count\_executions) AS [Execution Cnt]

,CONVERT(INT,SUM(rs.count\_executions \*

(rs.avg\_logical\_io\_reads + avg\_logical\_io\_writes)) /

SUM(rs.count\_executions)) AS [Avg IO]

,CONVERT(INT,SUM(rs.count\_executions \*

(rs.avg\_logical\_io\_reads + avg\_logical\_io\_writes))) AS [Total

IO]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_cpu\_time) /

SUM(rs.count\_executions)) AS [Avg CPU]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_cpu\_time)) AS [Total

CPU]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_duration) /

SUM(rs.count\_executions)) AS [Avg Duration]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_duration))

AS [Total Duration]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_physical\_io\_reads) /

SUM(rs.count\_executions)) AS [Avg Physical Reads]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_physical\_io\_reads))

AS [Total Physical Reads]

,CONVERT(INT,SUM(rs.count\_executions \*

rs.avg\_query\_max\_used\_memory) /

SUM(rs.count\_executions)) AS [Avg Memory Grant Pages]

,CONVERT(INT,SUM(rs.count\_executions \*

rs.avg\_query\_max\_used\_memory))

AS [Total Memory Grant Pages]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_rowcount) /

SUM(rs.count\_executions)) AS [Avg Rows]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_rowcount)) AS [Total

Rows]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_dop) /

SUM(rs.count\_executions)) AS [Avg DOP]

,CONVERT(INT,SUM(rs.count\_executions \* rs.avg\_dop)) AS [Total DOP]

FROM

sys.query\_store\_query q WITH (NOLOCK)

JOIN sys.query\_store\_plan qp WITH (NOLOCK) ON

q.query\_id = qp.query\_id

JOIN sys.query\_store\_query\_text qt WITH (NOLOCK) ON

q.query\_text\_id = qt.query\_text\_id

JOIN sys.query\_store\_runtime\_stats rs WITH (NOLOCK) ON

qp.plan\_id = rs.plan\_id

JOIN sys.query\_store\_runtime\_stats\_interval rsi WITH (NOLOCK) ON

rs.runtime\_stats\_interval\_id = rsi.runtime\_stats\_interval\_id

WHERE

rsi.end\_time >= DATEADD(DAY,-1,GETDATE())

GROUP BY

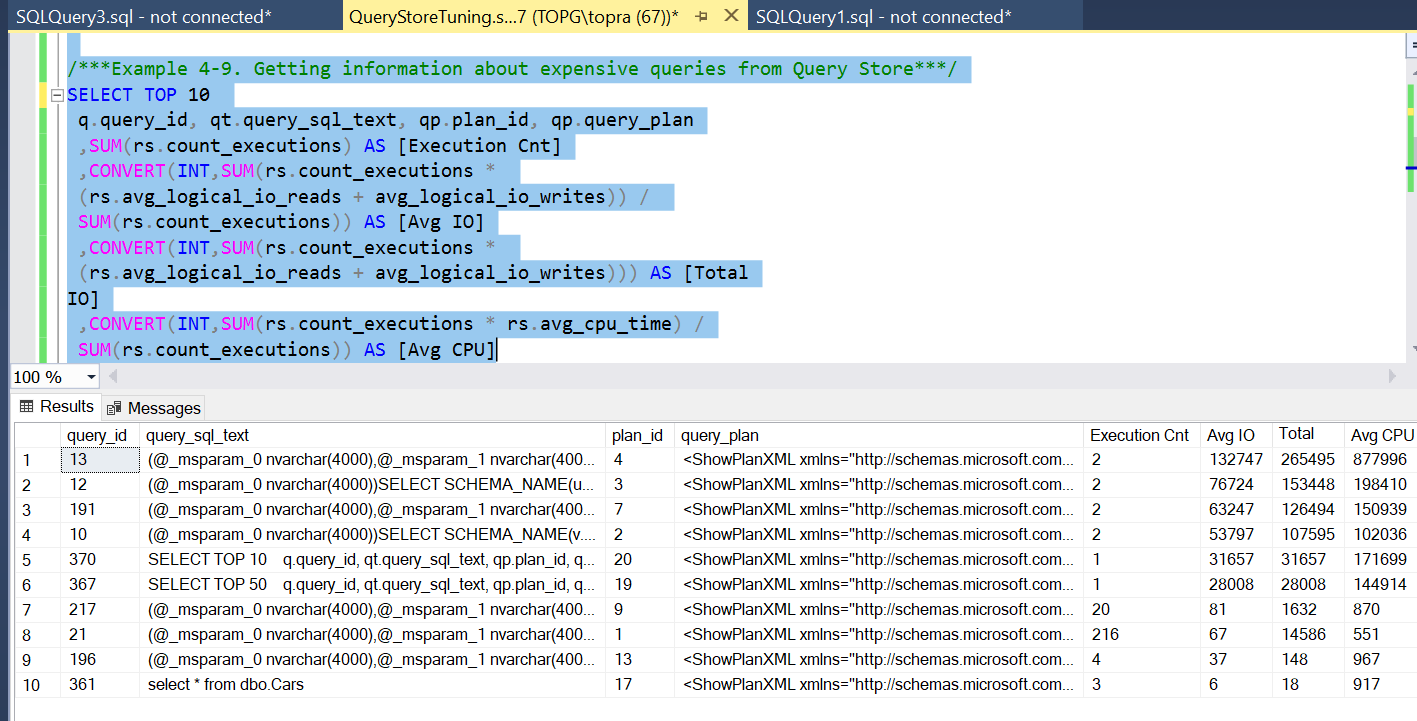
q.query\_id, qt.query\_sql\_text, qp.plan\_id, qp.query\_plan

ORDER BY

[Avg IO] DESC

OPTION (MAXDOP 1, RECOMPILE);

Output:



/\*\*Example 4-10. Queries with different context settings\*\*/

SELECT

q.query\_id, qt.query\_sql\_text

,COUNT(DISTINCT q.context\_settings\_id) AS [Context Setting Cnt]

,COUNT(DISTINCT qp.plan\_id) AS [Plan Count]

FROM

sys.query\_store\_query q WITH (NOLOCK)

JOIN sys.query\_store\_query\_text qt WITH (NOLOCK) ON

q.query\_text\_id = qt.query\_text\_id

JOIN sys.query\_store\_plan qp WITH (NOLOCK) ON

q.query\_id = qp.query\_id

GROUP BY

q.query\_id, qt.query\_sql\_text

HAVING

COUNT(DISTINCT q.context\_settings\_id) > 1

ORDER BY

COUNT(DISTINCT q.context\_settings\_id)

OPTION (MAXDOP 1, RECOMPILE);

/\*\*\*Example 4-11. Detecting queries with duplicated query\_hash value\*\*/

SELECT TOP 100

q.query\_hash

,COUNT(\*) AS [Query Count]

,AVG(rs.count\_executions) AS [Avg Exec Count]

FROM

sys.query\_store\_query q WITH (NOLOCK)

JOIN sys.query\_store\_plan qp WITH (NOLOCK) ON

q.query\_id = qp.query\_id

JOIN sys.query\_store\_runtime\_stats rs WITH (NOLOCK) ON

qp.plan\_id = rs.plan\_id

GROUP BY

q.query\_hash

HAVING

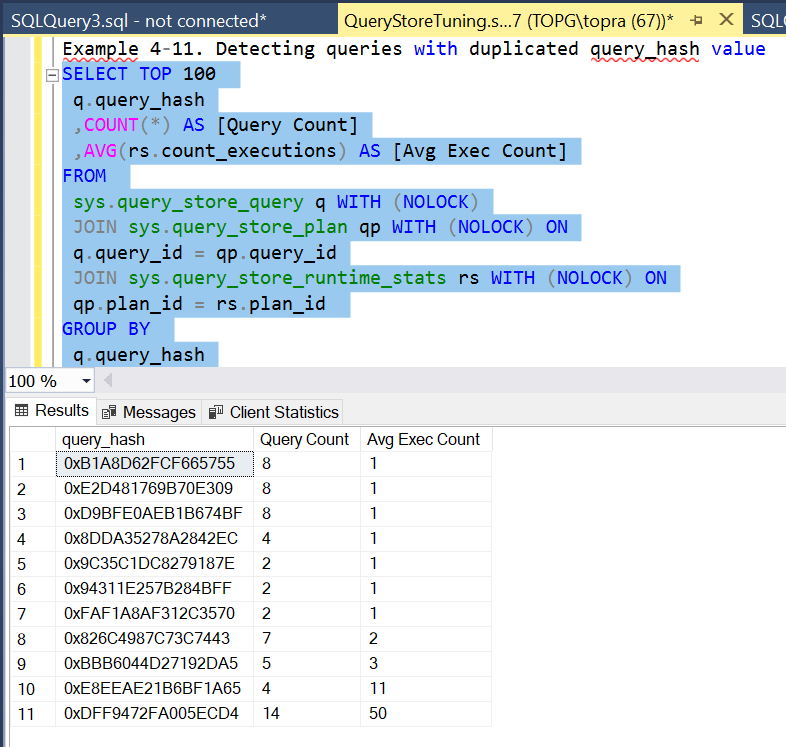
COUNT(\*) > 1

ORDER BY

[Avg Exec Count] ASC, [Query Count] DESC

OPTION(MAXDOP 1, RECOMPILE);

Output:



SQL Server keeps track of execution metrics for each cached plan and exposes them through the sys.dm\_exec\_query\_stats view. You can also get execution statistics for stored procedures, triggers, and scalar user-defined functions with sys.dm\_exec\_procedure\_stats, sys.dm\_exec\_trigger\_stats, and sys.dm\_exec\_function\_stats views, respectively.

# Chapter 5. Intro to Query Tuning

We have to understand the internal index structure and patterns that SQL Server uses to access data. This chapter thus begins with a high-level overview of B-Tree indexes and seek-and-scan operations.

Modern SQL Server versions support **three data storage and processing technologies**.

The oldest and most commonly used one is ***row-based storage***. With row-based storage, all table columns are combined together into the data rows that reside on 8KB data pages.

Starting with SQL Server 2012, you can store some indexes or entire tables in columnar format using ***column-based storage***. The data in such indexes is heavily compressed and stored on a per-column basis.

Finally, starting with SQL Server 2014, you can use In-Memory OLTP and store data in ***memory-optimized tables***. The data in such tables resides completely in memory and is great for heavy OLTP workloads.

Row-based storage is the default and by far most common storage technology in SQL Server. The CREATE TABLE and CREATE INDEX statements will store data in a row-based format unless you specify otherwise. It can handle moderate OLTP and analytical workloads and introduces less database administration overhead than column-store indexes and In-Memory OLTP.

## B-tree indexes

Clustered and non-clustered indexes have a very similar internal format called B-Tree. Let’s create an example table called Customers, defined in Listing 5-1. The table has the clustered index defined on CustomerId and non-clustered index on Name columns.

**Example 5-1. Customers table**

CREATE TABLE dbo.Customers

(

CustomerId INT NOT NULL,

Name NVARCHAR(64) NOT NULL,

Phone VARCHAR(32) NULL,

/\* Other Columns \*/

);

CREATE UNIQUE CLUSTERED INDEX IDX\_Customers\_CustomerId

ON dbo.Customers(CustomerId);

CREATE NONCLUSTERED INDEX IDX\_Customers\_Name

ON dbo.Customers(Name);

In the context of SQL databases, a SARGable (Search ARGument ABLE) query is a query that can use indexes to find the rows that match the search criteria efficiently. A non-SARGable query is a query that cannot use indexes efficiently to find the rows that match the search criteria and must perform a full table scan to find the rows.

You can analyze index fragmentation in the system with

***sys.dm\_db\_index\_physical\_stats*** view. The three most important columns

from the result set are:

1. avg\_page\_space\_used\_in\_percent

avg\_page\_space\_used\_in\_percent shows the average percentage of the

data storage space used on the page. This value shows you the internal

index fragmentation.

1. avg\_fragmentation\_in\_percent

avg\_fragmentation\_in\_percent provides you with information about

external index fragmentation. For tables with clustered indexes, it

indicates the percent of out-of-order pages when the next physical page

allocated in the index is different from the page referenced by the nextpage pointer of the current page. For heap tables, it indicates the percent

of out-of-order extents, when extents are not residing continuously in

data files.

1. fragment\_count

fragment\_count indicates how many continuous data fragments the

index has. Every fragment constitutes the group of extents adjacent to

each other. Adjacent data increases the chances that SQL Server will

use sequential I/O and Read-Ahead while accessing the data.

There are two index maintenance methods that ***reduce fragmentation***: index reorganize and index rebuild. Let’s look at each in turn.

Index reorganize

Index reorganize, often called index defragmentation, reorders leaf-level data pages into their logical order. It also tries to compress pages, reducing their internal fragmentation. This is an online operation that can be interrupted at any time without losing the operation’s progress up to the point of interruption. You can also reorganize indexes with the ALTER INDEX REORGANIZE command.

Index rebuild

Index rebuild (ALTER INDEX REBUILD), on the other hand, creates another copy of the index in the table. It is an offline operation, which will lock the table in non-Enterprise editions of SQL Server. In the Enterprise edition it can be done online, though it will still require a short table-level lock at the beginning and end of execution. Microsoft documentation recommends rebuilding indexes if their external fragmentation (*avg\_fragmentation\_in\_percent)* exceeds 30% and reorganize indexes for fragmentation between 5% and 30%. You can use those values as a rule of thumb; however, as I mentioned, it may be better to analyse and tune for your own use-cases.

Finally, in heap tables*, sys.dm\_db\_index\_physical\_stats* view provides the information about forwarding pointers with the *forwarded\_record\_count* column.

The number of I/O operations can be calculated based on the following formula:

(# of levels in nonclustered index) + (number of pages read from the leaf

level of nonclustered index) + (number of rows found) \* (# of levels in clustered index)

Technically, SQL Server can access the row in a heap though the single read operation; however, it is still expensive. Moreover, if the new version of the row does not fit into the old data page during an update, SQL Server will move it to another place, referencing it through another structure called *forwarding pointer*, which contains the address of the new (updated) version of the row.

# Index Fragmentation

There are two kinds of index fragmentation:

1. Internal
2. External

SQL Server always maintains the order of the data in the index, inserting new rows on the data pages to which they belong. If the data page does not have enough free space, SQL Server allocates a new page and places the row there, adjusting the pointers in the double-linked page list to maintain logical sorting order in the index. This operation is called page split, and it leads to index fragmentation, as you’ll see in this section.

# Statistics and Cardinality Estimation

SQL Server stores information about data distribution in the index in internal objects called statistics. By default, SQL Server creates statistics for each index in the database and uses it during query optimization.

**Density vector**, contains information about density for the combination of key values from the statistics (index). It is calculated based on the formula (1 / number of distinct values), and it indicates how many rows on average every combination of key values has.

SQL Server uses statistics information during query optimization estimating the number of rows that each operator in the execution plan would process and return to the next operator there. That process is called cardinality estimation.

**Listing 5-2 creates a table with clustered and nonclustered indexes and**

**populates it with some data. Finally, it provides information about the**

**statistics using the DBCC SHOW\_STATISTICS command.**

*Example 5-2. Examining statistics*

CREATE TABLE dbo.DBObjects

(

ID INT NOT NULL IDENTITY(1,1),

Name SYSNAME NOT NULL,

CreateDate DATETIME NOT NULL

);

CREATE UNIQUE CLUSTERED INDEX IDX\_DBObjects\_ID

ON dbo.DBObjects(ID);

INSERT INTO dbo.DBObjects(Name,CreateDate)

SELECT name, create\_date FROM sys.objects ORDER BY name;

-- Creating some duplicate values

INSERT INTO dbo.DBObjects(Name, CreateDate)

SELECT t1.Name, t1.CreateDate

FROM dbo.DBObjects t1 CROSS JOIN dbo.DBObjects t2

WHERE t1.ID = 5 AND t2.ID between 1 AND 20;

CREATE NONCLUSTERED INDEX IDX\_DBObjects\_Name\_CreateDate

ON dbo.DBObjectsName, CreateDate);

DBCC

SHOW\_STATISTICS('dbo.DBObjects','IDX\_DBObjects\_Name\_CreateDate');

## Statistics Maintenance

SSMS has another very useful feature called Live Query Statistics. This feature allows you to monitor query execution in runtime, detecting possible inefficiencies in the execution plan.

There is another useful new function, sys.dm\_exec\_query\_statistics\_xml, that utilizes lightweight profiling. It provides an in-flight execution plan for the currently running request.

## Example 5-5. Using sys.dm\_exec\_query\_statistics\_xml

SELECT

er.session\_id

,er.request\_id

,DB\_NAME(er.database\_id) as [database]

,er.start\_time

,CONVERT(DECIMAL(21,3),er.total\_elapsed\_time / 1000.) AS

[duration]

,er.cpu\_time

,SUBSTRING(

qt.text,

(er.statement\_start\_offset / 2) + 1,

((CASE er.statement\_end\_offset

WHEN -1 THEN DATALENGTH(qt.text)

ELSE er.statement\_end\_offset

END - er.statement\_start\_offset) / 2) + 1

) AS [statement]

,er.status

,er.wait\_type

,er.wait\_time

,er.wait\_resource

,er.blocking\_session\_id

,er.last\_wait\_type

,er.reads

,er.logical\_reads

,er.writes

,er.granted\_query\_memory

,er.dop

,er.row\_count

,er.percent\_complete

,es.login\_time

,es.original\_login\_name

,es.host\_name

,es.program\_name

,c.client\_net\_address

,ib.event\_info AS [buffer]

,qt.text AS [sql]

,p.query\_plan

FROM

sys.dm\_exec\_requests er WITH (NOLOCK)

OUTER APPLY sys.dm\_exec\_input\_buffer(er.session\_id,

er.request\_id) ib

OUTER APPLY sys.dm\_exec\_sql\_text(er.sql\_handle) qt

OUTER APPLY

sys.dm\_exec\_query\_statistics\_xml(er.session\_id) p

LEFT JOIN sys.dm\_exec\_connections c WITH (NOLOCK)

ON

er.session\_id = c.session\_id

LEFT JOIN sys.dm\_exec\_sessions es WITH (NOLOCK) ON

er.session\_id = es.session\_id

WHERE

er.status <> 'background'

AND er.session\_id > 50

ORDER BY

er.cpu\_time desc

OPTION (RECOMPILE, MAXDOP 1);

## Common Issues and Inefficiencies

1. Inefficient Code

With exception of black-box optimizations, where you don’t have access to the code, you should start the tuning by reviewing and, potentially, refactoring the queries. There are several anti-patterns and issues to detect.

## Non-SARGable predicates

## User-Defined Functions (UDF)

## Temporary Tables & Table Variables

Temporary tables and table variables are very valuable during query optimization. You can use them to persist the intermediate results of queries. This allows you to simplify queries, which may improve cardinality estimations and generate more efficient execution plans.

Both objects rely on tempdb. Even though table variables are slightly more efficient than temporary tables. Temporary tables, on the other hand, behave like regular tables. They maintain index statistics and allow SQL Server to use them during optimization.

Let’s look at a simple example: we’ll create a table and populate it with

some data. Then we’ll run two SELECT statements – with and without a

WHERE clause — as shown in Listing 5-6.

**Example 5-6. Index seek inefficiency**

CREATE TABLE dbo.T1

(

IndexedCol INT NOT NULL,

NonIndexedCol INT NOT NULL

);

CREATE UNIQUE CLUSTERED INDEX IDX\_T1

ON dbo.T1(IndexedCol);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 ROWS

,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 ROWS

,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 ROWS

,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2) -- 256 ROWS

,N5(C) AS (SELECT 0 FROM N4 AS T1 CROSS JOIN N4 AS T2) -- 65,536

ROWS

,N6(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N5 AS T2) -- 1,048,576

ROWS

,IDs(ID) AS (SELECT ROW\_NUMBER() OVER (ORDER BY (SELECT NULL)) FROM

N6)

INSERT INTO dbo.T1(IndexedCol, NonIndexedCol)

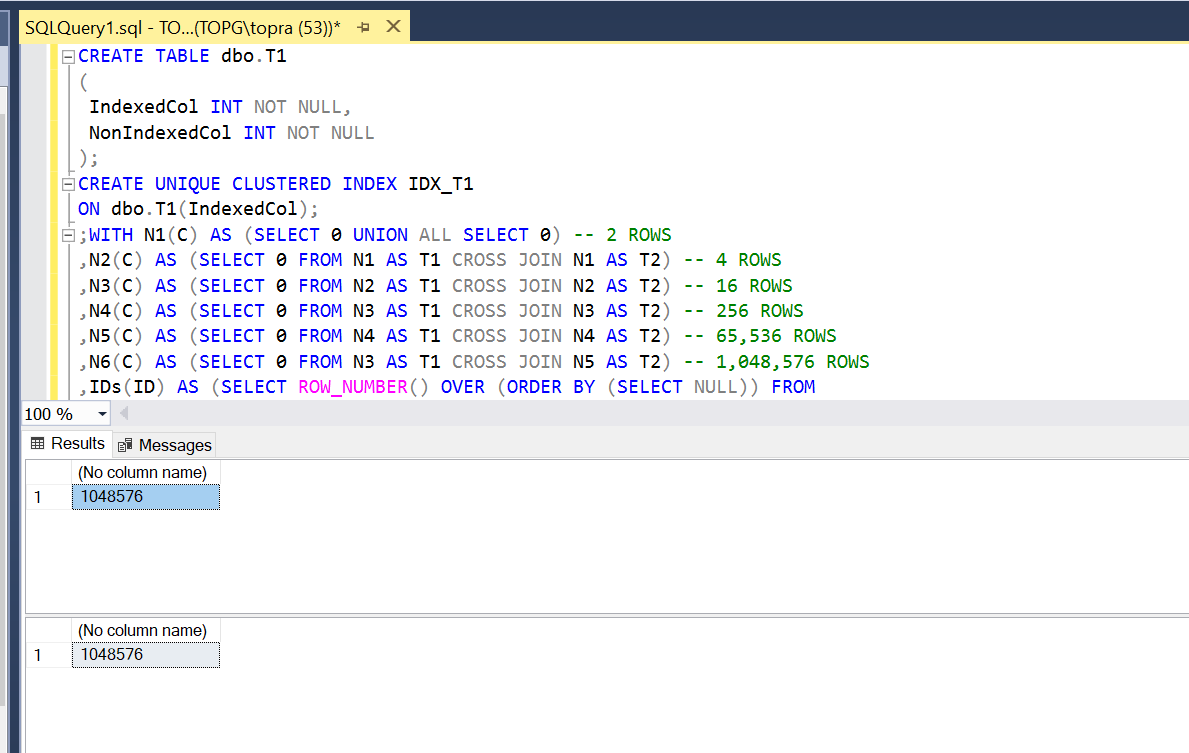
SELECT ID, ID FROM IDs;

SET STATISTICS IO ON

SELECT COUNT(\*) FROM dbo.T1;

SELECT COUNT(\*) FROM dbo.T1 WHERE IndexedCol > 0;

Output:



Incorrect JOIN TYPE

SQL Server uses many physical join operators during query execution. These belong to one of the three logical join types: loop, hash and merge.

Some of the physical join operators that SQL Server uses include:

1. Nested Loop Join: This operator reads each row from the first table, and then looks for a matching row in the second table using a nested loop. This operator is good for small tables or when the tables are already sorted on the join key.

Example 5-8. Loop join algorithm (pseudo code)

/\* Inner join \*/

for each row R1 in outer table

find row(s) R2 in inner table

if R1 joins with R2

return join (R1, R2)

/\* Outer join \*/

for each row R1 in outer table

find row(s) R2 in inner table

if R1 joins with R2

return join (R1, R2)

else

return join (R1, NULL)

1. Merge Join: This operator reads both tables and combines them into a single sorted stream. It then compares each row from each stream and returns the matching rows. This operator is good for larger tables or when the tables are already sorted on the join key.

Example 5-9. Inner merge join algorithm (pseudocode)

/\* Prerequirements: Inputs I1 and I2 are sorted \*/

get first row R1 from input I1

get first row R2 from input I2

while not end of either input

begin

if R1 joins with R2

begin

return join (R1, R2)

get next row R2 from I2

end

else if R1 < R2

get next row R1 from I1

else /\* R1 > R2 \*/

get next row R2 from I2

end

1. Hash Join: This operator builds a hash table in memory for one of the tables, and then probes the hash table with the rows from the other table to find the matching rows. This operator is good for large tables or when the tables are not sorted on the join key.

Example 5-10. Inner hash join algorithm (pseudocode)

/\* Build Phase \*/

for each row R1 in input I1

begin

calculate hash value on R1 join key

insert hash value to appropriate bucket in hash table

end

/\* Probe Phase \*/

for each row R2 in input I2

begin

calculate hash value on R2 join key

for each row R1 in hash table bucket

if R1 joins with R2

return join (R1, R2)

end

Loop join tends to perform better than hash join on small data sets, but hash join becomes more efficient as the size of the input increases. Merge join is most effective when the inputs are already sorted, but if they are not, it will add a sort operator to the process, which can be problematic with large amounts of data.

## Chapter 6. High CPU Load

Reducing I/O load through query tuning and code refactoring was usually enough to get the job done. Let’s talk about several common patterns that increase CPU load and options to address it. In conclusion, we will examine the advantages and disadvantages of using parallelism in systems and provide suggestions on how to adjust parallelism settings.

## Non-Optimized Queries and T-SQL Code

Why does your server have a high CPU load? There could be various reasons for a high CPU load on your server, but a common cause is poorly optimized queries. Even if the server has a fast disk subsystem and enough memory to cache all data, unoptimized queries can still increase the CPU load.

You can detect CPU-intensive queries using the techniques I discussed in Chapter 4, such as sorting data by CPU (worker) time while choosing targets for optimization. Optimizing those queries will decrease CPU load.

In SQL Server, the CPU (worker) time is the total amount of time that was spent by all worker threads on the CPU. It is a measure of how much CPU resources were used by the SQL Server process. This value can be useful for identifying performance bottlenecks, as high CPU usage may indicate that the server is overloaded or that the queries being run are not optimized. You can view the CPU worker time for a specific query by looking at the "worker time" column in the query execution plan or by using the sys.dm\_exec\_query\_stats dynamic management view.

NOTE: Reducing query duration will improve users’ experience, but I rarely choose optimization targets based on this factor. Optimizing queries with high resource usage usually reduces duration as well.

## Inefficient T-SQL Code

Inefficient T-SQL code also contributes to the problem. The benefits of properly designed and implemented T-SQL modules greatly overweight CPU overhead. But there’s one case we need to be more careful specifically – row-by-row processing.

Regardless of how you implement row-by-row processing – with cursors or with loops – it is inefficient. Imperative row-by-row execution will be slower and more CPU-intensive than declarative set-based logic.

To identify T-SQL statements that may be resource-intensive, you can use the sys.dm\_exec\_procedure\_stats, sys.dm\_exec\_function\_stats, and sys.dm\_exec\_trigger\_stats views to view execution statistics for T-SQL modules in the plan cache. Look for modules with high cumulative resource usage and examine their logic, paying particular attention to any row-by-row processing.

Take care when using CLR, external languages code, and extended stored procedures with complex logic, as they may be resource-intensive. Try to avoid making many function calls, especially with user-defined functions, as they can add overhead and may result in less efficient execution plans if they are not inlined. Additionally, be mindful of how views are used in your queries. If the database schema and definition are not optimized, views may cause unnecessary joins or access tables that are not needed, which can be especially problematic if there are no proper foreign keys defined.

## Scripts for Troubleshooting High CPU Load

A couple of scripts that are helpful when troubleshooting high CPU load. The first, in Listing 6-1, shows you CPU load on the server during the last 256 minutes.

Example 6-1. Getting CPU Load History

DECLARE

@now BIGINT;

SELECT @now = cpu\_ticks / (cpu\_ticks / ms\_ticks)

FROM sys.dm\_os\_sys\_info WITH (NOLOCK);

;WITH RingBufferData([timestamp], rec)

AS

(

SELECT [timestamp], CONVERT(XML, record) AS rec

FROM sys.dm\_os\_ring\_buffers WITH (NOLOCK)

WHERE

ring\_buffer\_type = N'RING\_BUFFER\_SCHEDULER\_MONITOR' AND

record LIKE N'%<SystemHealth>%'

)

,Data(id, SystemIdle, SQLCPU, [timestamp])

AS

(

SELECT

rec.value('(./Record/@id)[1]', 'int')

,rec.value

('(./Record/SchedulerMonitorEvent/SystemHealth/SystemIdle)

[1]','int')

,rec.value

('(./Record/SchedulerMonitorEvent/SystemHealth/ProcessUtilization)

[1]','int')

,[timestamp]

FROM RingBufferData

)

SELECT TOP 256

dateadd(MS, -1 \* (@now - [timestamp]), getdate()) AS [Event Time]

,SQLCPU AS [SQL Server CPU Utilization]

,SystemIdle AS [System Idle]

,100 - SystemIdle - SQLCPU AS [Other Processes CPU Utilization]

FROM Data

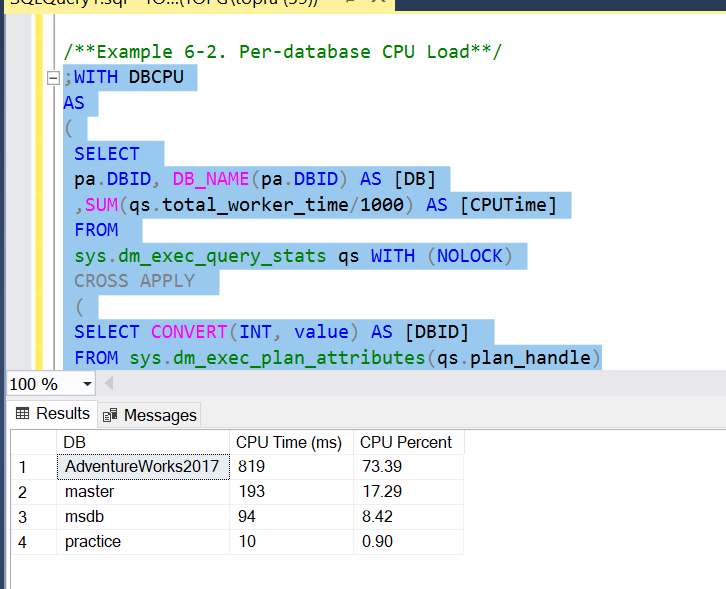
ORDER BY id desc

OPTION (RECOMPILE, MAXDOP 1);

Listing 6-2 is a script that can be used to analyze CPU usage per database. This can be useful if you have multiple databases on a server and want to see which ones are using the most CPU resources. It's important to note that this script uses plan cache data, so the results may not be completely accurate. This information may be helpful when deciding if you need to move some of the busiest databases to a different server.

l

Output:



## Non-Optimized Query Patterns to Watch For

In case of non-optimized queries, there are two distinct patterns that can trigger high CPU load. I call them “the worst offenders” and “death by a thousand cuts.”

(This terminology is by no means standard – it’s just how I like to differentiate between them.)

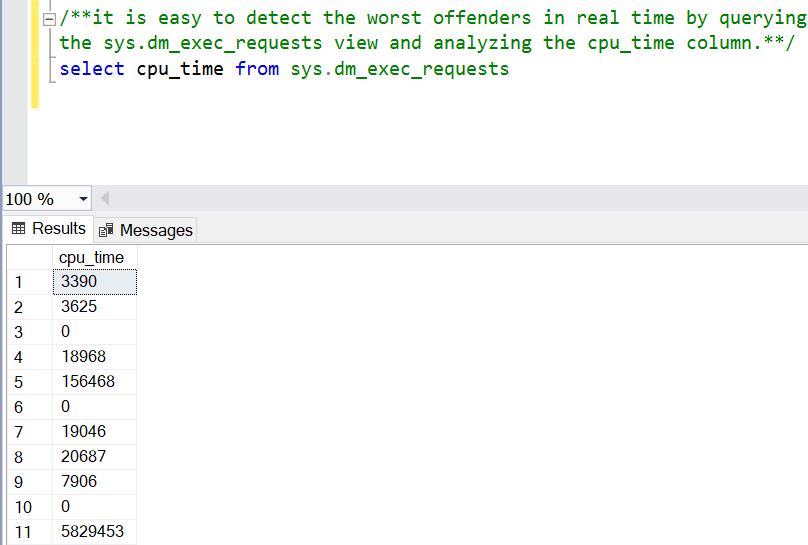
1. Worst Offender

/\*\*it is easy to detect the worst offenders in real time by querying

the sys.dm\_exec\_requests view and analyzing the cpu\_time column.\*\*/

#### select cpu\_time from sys.dm\_exec\_requests;

Output:



1. Death by a thousand cuts

The second pattern of server load, "death by a thousand cuts," occurs when a large number of requests are running simultaneously and causing a high load on the server. Each request may be optimized and relatively small, but the sheer number of requests can still strain the CPU and overall server performance. This type of load can be more difficult to address. Query Optimization may help solve the issues.

# Query Compilation and Plan Caching

When you submit a query to the database, SQL Server needs to compile and optimize it in order to execute it. This process can use up a lot of resources, so SQL Server tries to minimize the number of times it has to do this by storing execution plans in a cache so they can be used again later. This cache, called the plan cache, stores the plans for not just regular client queries and batches, but also for objects like stored procedures, triggers, and user-defined functions.

Caching and reusing plans can significantly decrease the number of compilations and the CPU usage. However, it can also lead to issues. We will examine some of the most frequent problems that occur, starting with the sensitivity to parameters in plans that are sensitive to parameters. (This is sometimes referred to as "parameter sniffing," which is a term that describes the behaviour of SQL Server that causes this problem).

Example 6-3. Parameter-sensitive plans: Table creation

select \* from dbo.Orders

CREATE TABLE dbo.Orders

(

OrderId INT NOT NULL IDENTITY(1,1),

OrderNum VARCHAR(32) NOT NULL,

CustomerId UNIQUEIDENTIFIER NOT NULL,

Amount MONEY NOT NULL,

StoreId INT NOT NULL,

Fulfilled BIT NOT NULL

);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 rows

,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 rows

,N3(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N2 AS T2) -- 16 rows

,N4(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N3 CROSS JOIN N2 AS T3)

-- 1024 rows

,N5(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N4 AS T2 ) -- 1,048,576rows

,IDs(ID) AS (SELECT ROW\_NUMBER() OVER (ORDER BY (SELECT NULL)) FROM N5)

INSERT INTO dbo.Orders(OrderNum, CustomerId, Amount, StoreId, Fulfilled)

select

'Order: ' + convert(varchar(32),ID)

,newid()

,ID % 100

,ID % 10

,1

from IDs;

INSERT INTO dbo.Orders(OrderNum, CustomerId, Amount, StoreId,

Fulfilled)

select top 10 OrderNum, CustomerId, Amount, 99, 0

from dbo.Orders

order by OrderId;

CREATE UNIQUE CLUSTERED INDEX IDX\_Orders\_OrderId

ON dbo.Orders(OrderId);

CREATE NONCLUSTERED INDEX IDX\_Orders\_CustomerId

ON dbo.Orders(CustomerId);

CREATE NONCLUSTERED INDEX IDX\_Orders\_StoreId

ON dbo.Orders(StoreId);

Listing 6-4. Parameter-sensitive plans: Stored procedure

CREATE PROC dbo.GetTotalPerStore(@StoreId int)

AS

SELECT SUM(Amount) as [Total Amount]

FROM dbo.Orders

WHERE StoreId = @StoreId;

Example 6-5. Parameter-sensitive plans: Calling the procedure

EXEC dbo.GetTotalPerStore @StoreId = 5;

EXEC dbo.GetTotalPerStore @StoreId = 99;

In SQL Server, "parameterization" is the process of replacing constant values in a SQL statement with parameters, in order to prevent SQL injection attacks.

Example 6-6. Parameter-sensitive plans: Calling the procedure (Test 2)

DBCC FREEPROCCACHE;

EXEC dbo.GetTotalPerStore @StoreId = 99;

EXEC dbo.GetTotalPerStore @StoreId = 5;

If you are using SQL Server 2017 or above, you can benefit from automatic plan correction, which is part of the automatic tuning technology. When this feature is enabled, SQL Server can detect parameter sniffing issues and automatically force the last known good plan that was used before regression occurred. Automatic plan correction relies on the Force Plan feature of Query Store and, as you can guess, requires Query Store to be enabled in the database. Moreover, you need to enable it in the database with ALTER DATABASE SET AUTOMATIC\_TUNING (FORCE\_LAST\_GOOD\_PLAN = ON) statement. You can read more about it in the Microsoft documentation.

If neither of those options works, you can force the recompilation of either stored procedure using EXECUTE WITH RECOMPILE or a statement-level recompile with OPTION (RECOMPILE) clauses. Listing 6-7 shows the latter approach.

Example 6-7. Parameter-sensitive plans: Statement-level recompile

ALTER PROC dbo.GetTotalPerStore(@StoreId int)

AS

SELECT SUM(Amount) as [Total Amount]

FROM dbo.Orders

WHERE StoreId = @StoreId

OPTION (RECOMPILE);

GO

EXEC dbo.GetTotalPerStore @StoreId = 99;

EXEC dbo.GetTotalPerStore @StoreId = 5;

Parameter sniffing refers to the process of creating a query plan based on the parameters passed to a query when it is first executed, and then reusing that plan for subsequent executions of the query, even if the parameters passed in are different.

Here is a simple example of parameter sniffing:

Let's say you have a query that retrieves data from a table based on a specific value of a column. The query is executed frequently with different values for the column. The first time the query is executed, it uses the value "A" as the parameter. The query optimizer creates a query plan based on this value and the distribution of data in the table. The plan is stored in the cache, and is reused for subsequent executions of the query with different parameter values.

DECLARE @param VARCHAR(10) = 'A';

SELECT \*

FROM my\_table

WHERE column = @param;

However, let's say that most of the time the query is executed with a different value, "B". The distribution of data for value "B" is very different from the distribution of data for value "A", and the query plan created for value "A" is not efficient for value "B". As a result, the query performs poorly when it is executed with value "B".

This is an example of parameter sniffing, because the query optimizer is using the same query plan for different parameter values, even though the plan is not optimal for all of the values. To resolve this issue, you can use the OPTIMIZE FOR hint, or use local variables or dynamic SQL so that the query optimizer will create a new plan each time the query is executed.

This can be useful in situations where parameter sniffing is causing performance issues. It allows you to force the optimizer to create a plan that is optimized for a specific value, rather than for the first value that was used when the query was first run.

Simple Parameterization

Forced Parameterization

## Chapter 7. Troubleshooting Memory Issues

I’d like to repeat a few things here. Memory is the key resource in SQL Server. Adding more memory to the servers is often the fastest and cheapest way to improve system performance.

Listing 7-1. Analyzing OS and SQL Server memory usage

SELECT total\_physical\_memory\_kb / 1024 AS [Physical Memory (MB)]

,available\_physical\_memory\_kb / 1024 AS [Available Memory (MB)]

,total\_page\_file\_kb / 1024 AS [Page File Commit Limit (MB)]

,available\_page\_file\_kb / 1024 AS [Available Page File (MB)]

,(total\_page\_file\_kb - total\_physical\_memory\_kb) / 1024

AS [Physical Page File Size (MB)]

,system\_cache\_kb / 1024 AS [System Cache (MB)]

/\* Values: LOW/HIGH/STEADY \*/

,system\_memory\_state\_desc AS [System Memory State]

FROM sys.dm\_os\_sys\_memory WITH (NOLOCK);

SELECT

physical\_memory\_in\_use\_kb / 1024

AS [SQL Server Memory Usage (MB)]

,locked\_page\_allocations\_kb / 1024

AS [SQL Server Locked Pages Allocation (MB)]

,large\_page\_allocations\_kb / 1024

AS [SQL Server Large Pages Allocation (MB)]

,memory\_utilization\_percentage

,available\_commit\_limit\_kb

,process\_physical\_memory\_low /\* May indicate memory pressure \*/

,process\_virtual\_memory\_low

FROM sys.dm\_os\_process\_memory WITH (NOLOCK);

Listing 7-2. sp\_server\_diagnostics\_component\_result event in system\_health session (partial)

<resource lastNotification="RESOURCE\_MEM\_STEADY"

outOfMemoryExceptions="0" isAnyPoolOutOfMemory="0"

processOutOfMemoryPeriod="0">

<memoryReport name="Process/System Counts" unit="Value">

<entry description="Available Physical Memory"

value="65669554176" />

<entry description="Available Virtual Memory"

value="138792447782912" />

<entry description="Available Paging File" value="67695706112"

/>

<..>

</memoryReport>

<memoryReport name="Memory Manager" unit="KB">

<entry description="Locked Pages Allocated" value="641593188" />

<entry description="Large Pages Allocated" value="3248128" />

<entry description="Target Committed" value="653261832" />

<entry description="Current Committed" value="653263320" />

<..>

</memoryReport>

</resource>

There are no limitations on how much memory SQL Server can utilize with Enterprise Edition. Add as much memory as your server can support and use the fastest memory possible.

In Standard Edition, the buffer pool size is limited to 128GB, but you’ll need additional memory for other SQL Server components and the OS.

I recommend provisioning the servers with 192GB of RAM to be on the safe side. You will need even more memory if you are using column-store indexes or In-Memory OLTP.

You can improve memory utilization by decreasing internal index fragmentation (the avg\_page\_space\_used\_in\_percent column in the sys.dm\_db\_index\_physical\_stats view) and/or applying data compression.

The last major component in SQL Server dynamic memory management is called the memory broker. Memory brokers supervise memory clerks by adjusting their memory usage based on available process memory, memory pressure, and other conditions.

You can look at the sys.dm\_os\_memory\_brokers view to see memory broker state and amount of memory they allocate.

Analyzing memory usage by looking at memory clerks using the sys.dm\_os\_memory\_clerks view. Listing 7-3 shows how to do that.

Listing 7-3. Analyzing memory usage

SELECT TOP 15

[type] AS [Memory Clerk]

,CONVERT(DECIMAL(16,3),SUM(pages\_kb) / 1024.0) AS [Memory

Usage(MB)]

FROM sys.dm\_os\_memory\_clerks WITH (NOLOCK)

GROUP BY [type]

ORDER BY sum(pages\_kb) DESC

Listing 7-4. Buffer pool usage on per-database basis

;WITH BufPoolStats

AS

(

SELECT

database\_id

,COUNT\_BIG(\*) AS page\_count

,CONVERT(DECIMAL(16,3),COUNT\_BIG(\*) \* 8 / 1024.) AS size\_mb

,AVG(read\_microsec) AS avg\_read\_microsec

FROM

sys.dm\_os\_buffer\_descriptors WITH (NOLOCK)

GROUP BY

database\_id

)

SELECT

DB\_NAME(database\_id) AS [DB]

,size\_mb

,page\_count

,avg\_read\_microsec

,CONVERT(DECIMAL(5,2), 100. \* (size\_mb / SUM(size\_mb) OVER()))

AS [Percent]

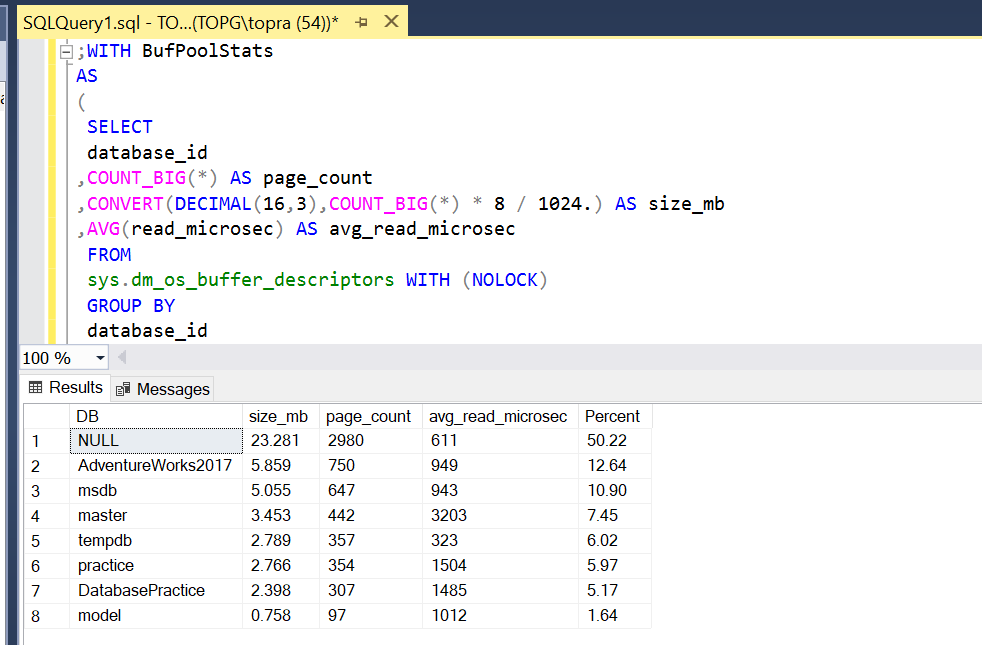
FROM

BufPoolStats

ORDER BY

size\_mb DESC

OPTION (MAXDOP 1, RECOMPILE);



Listing 7-5. Running 1,000 ad-hoc queries and examining plan cache content.

DBCC FREEPROCCACHE

GO

DECLARE

@SQL NVARCHAR(MAX)

,@I INT = 0

WHILE @I < 1000

BEGIN

SELECT @SQL =

N'DECLARE @C INT;SELECT @C=object\_id FROM sys.objects WHERE

object\_id='

+ CONVERT(NVARCHAR(10),@I);

EXEC(@SQL);

SELECT @I += 1;

END;

SELECT

p.usecounts, p.cacheobjtype, p.objtype, p.size\_in\_bytes, t.

[text]

FROM

sys.dm\_exec\_cached\_plans p WITH (NOLOCK)

CROSS APPLY sys.dm\_exec\_sql\_text(p.plan\_handle) t

WHERE

p.objtype = 'Adhoc'

ORDER BY

p.objtype DESC

OPTION (RECOMPILE);

SELECT

CONVERT(DECIMAL(12,3),SUM(1. \* p.size\_in\_bytes)/1024.) AS [Size

(KB)]

FROM

sys.dm\_exec\_cached\_plans p WITH (NOLOCK)

WHERE

p.objtype = 'Adhoc'

OPTION (RECOMPILE);

Enable the AD-Hoc Workload, GOTO instance server right-click-properties-advanced- search for Ad-hoc – click to true.

Listing 7-6. Running parameterized query and examine plan cache content

DBCC FREEPROCCACHE

GO

DECLARE

@SQL NVARCHAR(MAX),@I INT = 0

WHILE @I < 1000

BEGIN

SELECT @SQL =

N'DECLARE @C INT;SELECT @C=object\_id FROM sys.objects WHERE

object\_id=@P';

EXEC sp\_executesql @SQL=@SQL,@Params=N'@P INT',@P = @I;

SELECT @I += 1;

END;

SELECT

p.usecounts, p.cacheobjtype, p.objtype, p.size\_in\_bytes, t.

[text]

FROM

sys.dm\_exec\_cached\_plans p WITH (NOLOCK)

CROSS APPLY sys.dm\_exec\_sql\_text(p.plan\_handle) t

WHERE

p.objtype = 'Adhoc'

ORDER BY

p.objtype DESC

OPTION (RECOMPILE);

SELECT

CONVERT(DECIMAL(12,3),SUM(1. \* p.size\_in\_bytes)/1024.) AS [Size

(KB)]

FROM

sys.dm\_exec\_cached\_plans p WITH (NOLOCK)

WHERE

p.objtype = 'Adhoc'

OPTION (RECOMPILE);

Listing 7-7. Analyzing plan cache

-- Number of cached object and their memory usage grouped by type

SELECT

cacheobjtype

,objtype

,COUNT(\*) AS [Count]

,CONVERT(DECIMAL(12,3),SUM(1.\*size\_in\_bytes)/1024./1024.)

AS [Size (MB)]

FROM

sys.dm\_exec\_cached\_plans WITH (NOLOCK)

GROUP BY

cacheobjtype, objtype

ORDER BY

[Size (MB)] DESC

OPTION (RECOMPILE);

-- Statistics on single-used execution plans

SELECT

COUNT(\*) AS [Single-used plan count]

,CONVERT(DECIMAL(10,3),SUM(cp.size\_in\_bytes)/1024./1024.)

AS [Size (MB)]

FROM

sys.dm\_exec\_cached\_plans cp WITH (NOLOCK)

WHERE

cp.objtype in (N'Adhoc', N'Prepared') AND

cp.usecounts = 1

OPTION (RECOMPILE);

-- 25 most memory-intensive single-used plans

SELECT TOP 25

DB\_NAME(t.dbid) as [DB]

,cp.usecounts

,cp.plan\_handle

,t.[text]

,cp.objtype

,cp.size\_in\_bytes

,CONVERT(DECIMAL(12,3),cp.size\_in\_bytes/1024.) as [Size (KB)]

FROM

sys.dm\_exec\_cached\_plans cp WITH (NOLOCK)

CROSS APPLY sys.dm\_exec\_sql\_text(cp.plan\_handle) t

WHERE

cp.cacheobjtype = N'Compiled Plan'

AND cp.objtype in (N'Adhoc', N'Prepared')

AND cp.usecounts = 1

ORDER BY

cp.size\_in\_bytes DESC

OPTION (RECOMPILE);

Listing 7-8. Memory grant information

SELECT mg.session\_id

,t.text AS [sql]

,qp.query\_plan AS [plan]

,mg.is\_small /\* Resource Semaphore Queue information \*/

,mg.dop

,mg.query\_cost

,mg.request\_time

,mg.grant\_time

,mg.wait\_time\_ms

,mg.required\_memory\_kb

,mg.requested\_memory\_kb

,mg.granted\_memory\_kb

,mg.used\_memory\_kb

,mg.max\_used\_memory\_kb

,mg.ideal\_memory\_kb

FROM

sys.dm\_exec\_query\_memory\_grants mg WITH (NOLOCK)

CROSS APPLY sys.dm\_exec\_sql\_text(mg.sql\_handle) t

CROSS APPLY sys.dm\_exec\_query\_plan(mg.plan\_handle) qp

--WHERE -- Uncomment to see only pending memory grants

-- mg.grant\_time IS NULL

ORDER BY

mg.requested\_memory\_kb DESC

OPTION (RECOMPILE, MAXDOP 1);

Listing 7-9. Optimizing memory intensive queries: Table creation

CREATE TABLE dbo.Orders

(

OrderID INT NOT NULL,

OrderDate DATETIME2(0) NOT NULL,

Placeholder CHAR(8000) NULL,

CONSTRAINT PK\_Orders PRIMARY KEY CLUSTERED(OrderID)

);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 ROWS

,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 ROWS

,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 ROWS

,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2) -- 256

ROWS

,N5(C) AS (SELECT 0 FROM N4 AS T1 CROSS JOIN N4 AS T2) -- 65,536

ROWS

,IDs(ID) AS (SELECT ROW\_NUMBER() OVER (ORDER BY (SELECT NULL))

FROM N5)

INSERT INTO dbo.Orders(OrderID, OrderDate)

SELECT ID, DATEADD(day,ID % 365, '2021-01-01')

FROM IDs;

Listing 7-10. Optimizing memory intensive queries: Test query 1

SELECT TOP 200 OrderID, OrderDate, Placeholder

FROM dbo.Orders

ORDER BY OrderDate DESC

Listing 7-11. Optimizing memory intensive queries: Test query 2

SELECT TOP 200 OrderID, OrderDate, Placeholder

FROM dbo.Orders

WHERE OrderDate BETWEEN '2021-07-01' AND '2021-08-01'

ORDER BY Placeholder;

Listing 7-12. Optimizing memory intensive queries: Outdating statistics

ALTER INDEX IDX\_Orders\_OrderDate ON dbo.Orders

SET (STATISTICS\_NORECOMPUTE = ON);

DELETE FROM dbo.Orders

WHERE OrderDate BETWEEN '2021-07-02' AND '2021-09-01';

DBCC FREEPROCCACHE;

Listing 7-13. Hypothetical query

SELECT Col1, Col2

FROM T1

ORDER BY Col3

OPTION(MIN\_GRANT\_PERCENT=0.5,MAX\_GRANT\_PERCENT=3);

Listing 7-14. Limiting amount of memory for In-Memory OLTP

CREATE RESOURCE POOL InMemoryDataPool

WITH (MIN\_MEMORY\_PERCENT=40,MAX\_MEMORY\_PERCENT=40);

ALTER RESOURCE GOVERNOR RECONFIGURE;

EXEC sys.sp\_xtp\_bind\_db\_resource\_pool

@database\_name = 'InMemoryOLTPDemo'

,@pool\_name = 'InMemoryDataPool';

-- You need to take DB offline and bring it back online

-- for the changes to take effect

ALTER DATABASE MyDB SET OFFLINE;

ALTER DATABASE MyDB SET ONLINE;

MEMORYCLERK\_SQLCLR, MEMORYCLERK\_SQLCLRASSEMBLY and MEMORYCLERK\_SQLEXTENSIBILITY clerks are used for memory allocations in CLR and other supported language extensions (Java, R, and Python).

Listing 7-15. Analyzing memory consumption of memory-optimized tables

SELECT

ms.object\_id

,s.name + '.' + t.name AS [table]

,ms.memory\_allocated\_for\_table\_kb

,ms.memory\_used\_by\_table\_kb

,ms.memory\_allocated\_for\_indexes\_kb

,ms.memory\_used\_by\_indexes\_kb

FROM sys.dm\_db\_xtp\_table\_memory\_stats ms WITH (NOLOCK)

LEFT OUTER JOIN sys.tables t WITH (NOLOCK) ON

ms.object\_id = t.object\_id

LEFT OUTER JOIN sys.schemas s WITH (NOLOCK) ON

t.schema\_id = s.schema\_id

ORDER BY

ms.memory\_allocated\_for\_table\_kb DESC;

Listing 7-16 shows the code that detects the ten oldest In-Memory OLTP transactions. You can use it for troubleshooting and you can build monitoring and alerting around it

Listing 7-16. Detecting 10 oldest In-Memory OLTP transactions

SELECT TOP 10

t.session\_id

,t.transaction\_id

,t.begin\_tsn

,t.end\_tsn

,t.state\_desc

,t.result\_desc

,SUBSTRING(

qt.text

,er.statement\_start\_offset / 2 + 1

,(CASE er.statement\_end\_offset

WHEN -1 THEN datalength(qt.text)

ELSE er.statement\_end\_offset

END - er.statement\_start\_offset

) / 2 +1

) AS SQL

FROM

sys.dm\_db\_xtp\_transactions t WITH (NOLOCK)

LEFT OUTER JOIN sys.dm\_exec\_requests er ON

t.session\_id = er.session\_id

CROSS APPLY sys.dm\_exec\_sql\_text(er.sql\_handle) qt

WHERE

t.state IN (0,3) /\* ACTIVE/VALIDATING \*/

ORDER BY

t.begin\_tsn

OPTION (RECOMPILE, MAXDOP 1);

You can analyze the current memory usage by looking at the memory consumption of various memory clerks. Detect anomalies and address rootcauses of the issues.

Monitor the status of memory grants and presence of RESOURCE\_SEMAPHORE waits. Optimize queries with high memory grants when possible. Enable memory grant feedback feature if it is available in your version of SQL Server.

# Chapter 8. Troubleshooting TempDB Usage and Performance

The tempdb is the system database, which is shared across all user and system sessions. It stores user-created and internal temporary objects and data, and is used by many processes. High tempdb performance and throughput are essential for good server performance.

After all, tempdb is just another database, and reducing its load usually improves its throughput.

Listing 9-1. Tempdb space usage

SELECT

CONVERT(DECIMAL(12,3),

SUM(user\_object\_reserved\_page\_count) / 128.)

AS [User Objects (MB)]

,CONVERT(DECIMAL(12,3),

SUM(internal\_object\_reserved\_page\_count) / 128.

) AS [Internal Objects (MB)]

,CONVERT(DECIMAL(12,3),

SUM(version\_store\_reserved\_page\_count) / 128.

) AS [Version Store (MB)]

,CONVERT(DECIMAL(12,3),

SUM(unallocated\_extent\_page\_count) / 128.

) AS [Free Space (MB)]

FROM

tempdb.sys.dm\_db\_file\_space\_usage WITH (NOLOCK);

There are two kind of **temporary tables** – global and local. They differ in lifespan and visibility.

***Global temporary*** tables are created with names that start with two ***hash symbols (##)*** and are visible to all sessions.

They are dropped when the session in which they were created disconnects and other sessions stop referencing them. You can use global temporary tables to store and share temporary data between sessions. This approach, however, is fragile and prone to errors.

***Local temporary*** tables are named starting with one ***hash symbol (#)*** and are visible only in the session in which they were created. When multiple sessions simultaneously create local temporary tables with the same name, every session will have its own instance of the table.

Listing 9-2. Cardinality estimations: Temporary table creation

CREATE TABLE #TT(ID INT NOT NULL PRIMARY KEY);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 rows

,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 rows

,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16

rows

,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2) -- 256

rows

,IDs(ID) AS (SELECT ROW\_NUMBER() OVER (ORDER BY (SELECT NULL))

FROM N4)

INSERT INTO #TT(ID)

SELECT ID FROM IDs;

Listing 9-3. Cardinality estimations: Selecting data from temporary objects

DECLARE

@TTV TABLE(ID INT NOT NULL PRIMARY KEY);

INSERT INTO @TTV(ID)

SELECT ID FROM #TT;

SELECT COUNT(\*) FROM #TT;

SELECT COUNT(\*) FROM @TTV;

SELECT COUNT(\*) FROM @TTV OPTION (RECOMPILE);

Listing 9-4. All rows in the tables have positive ID values.

Listing 9-4. Cardinality estimations: Selecting data with WHERE clause

DECLARE

@TTV TABLE(ID INT NOT NULL PRIMARY KEY);

INSERT INTO @TTV(ID)

SELECT ID FROM #TT;

SELECT COUNT(\*) FROM #TT WHERE ID > 0;

SELECT COUNT(\*) FROM @TTV WHERE ID > 0;

SELECT COUNT(\*) FROM @TTV WHERE ID > 0 OPTION (RECOMPILE);

Let’s look at Listing 9-5 and define the stored procedure that creates and

drops the temporary table.

Listing 9-5. Temporary object caching: Stored procedure

CREATE PROC dbo.TempTableCaching

AS

CREATE TABLE #T(C INT NOT NULL PRIMARY KEY);

DROP TABLE #T;

Listing 9-6. Temporary object caching: Running stored procedure

CHECKPOINT;

GO

EXEC dbo.TempTableCaching;

SELECT

Operation, Context, AllocUnitName

,[Transaction Name], [Description]

FROM

tempdb.sys.fn\_dblog(null, null);

Listing 9-7 shows how you can use table-valued parameters. (This is just an example of a possible usage scenario, not a reference implementation for any use cases!) It also demonstrates that table variables are not transaction-aware. You can use them to pass the information outside of the transaction you are rolling back.

Listing 9-7. Using table-valued parameters

CREATE TYPE dbo.tvpTransfers AS TABLE

(

FromAccount BIGINT NOT NULL,

ToAccount BIGINT NOT NULL,

ADate DATETIME2(0) NOT NULL,

Amount MONEY NOT NULL,

PRIMARY KEY(FromAccount,ToAccount)

);

GO

CREATE PROC dbo.ProcessRejectedTransfers

(

@RejectedTransfers dbo.tvpTransfers READONLY

)

AS

SELECT FromAccount, ToAccount, ADate, Amount

FROM @RejectedTransfers;

GO

CREATE PROC dbo.DoTransfers

(

@Transfers dbo.tvpTransfers READONLY

)

AS

DECLARE

@RejectedTransfers dbo.tvpTransfers

BEGIN TRAN

INSERT INTO @RejectedTransfers

(FromAccount, ToAccount, ADate, Amount)

SELECT FromAccount, ToAccount, ADate, Amount

FROM @Transfers

WHERE Amount > 10000;

/\* ... \*/

ROLLBACK -- Table variables are not transaction-aware.

EXEC sp\_executesql

N’EXEC dbo.ProcessRejectedTransfers @Transfers;’

,N’@Transfers dbo.tvpTransfers READONLY’

,@Transfers = @RejectedTransfers;

GO

DECLARE

@Transfers dbo.tvpTransfers

INSERT INTO @Transfers

(FromAccount, ToAccount, ADate, Amount)

VALUES

(1,2,'2021-08-01',100)

,(3,4,'2021-08-02',15000)

,(5,6,'2021-08-03',20000);

EXEC dbo.DoTransfers @Transfers;

Listing 9-8. Detecting the five oldest row-versioning transactions

SELECT TOP 5

at.transaction\_id

,at.elapsed\_time\_seconds

,at.session\_id

,s.login\_time

,s.login\_name

,s.host\_name

,s.program\_name

,s.last\_request\_start\_time

,s.last\_request\_end\_time

,er.status

,er.wait\_type

,er.wait\_time

,er.blocking\_session\_id

,er.last\_wait\_type

,st.text AS [SQL]

FROM

sys.dm\_tran\_active\_snapshot\_database\_transactions at WITH

(NOLOCK)

JOIN sys.dm\_exec\_sessions s WITH (NOLOCK) on

at.session\_id = s.session\_id

LEFT JOIN sys.dm\_exec\_requests er WITH (NOLOCK) on

at.session\_id = er.session\_id

CROSS APPLY

sys.dm\_exec\_sql\_text(er.sql\_handle) st

ORDER BY

at.elapsed\_time\_seconds DESC;

Listing 9-9. Version store usage per database

-- SQL Server 2016 SP2 and above

SELECT

DB\_NAME(database\_id) AS [DB]

,database\_id

,reserved\_page\_count

,CONVERT(DECIMAL(12,3),reserved\_space\_kb / 1024.)

AS [Reserved Space (MB)]

FROM

sys.dm\_tran\_version\_store\_space\_usage WITH (NOLOCK)

OPTION (RECOMPILE);

-- SQL Server 2014 and below. Less accurate.

SELECT

DB\_NAME(database\_id) AS [DB]

,database\_id

,CONVERT(DECIMAL(12,3),

SUM(record\_length\_first\_part\_in\_bytes +

record\_length\_second\_part\_in\_bytes) / 1024. / 1024.

) AS [Version Store (MB)]

FROM

sys.dm\_tran\_version\_store WITH (NOLOCK)

GROUP BY

database\_id

OPTION (RECOMPILE, MAXDOP 1);

Listing 9-10. Using xEvents to detect queries that spill to tempdb

CREATE EVENT SESSION [Spills]

ON SERVER

ADD EVENT sqlserver.hash\_warning

(

ACTION

(

sqlserver.database\_id

,sqlserver.plan\_handle

,sqlserver.session\_id

,sqlserver.sql\_text

,sqlserver.query\_hash

,sqlserver.query\_plan\_hash

)

WHERE ([sqlserver].[is\_system]=0)

),

ADD EVENT sqlserver.sort\_warning

(

ACTION

(

sqlserver.database\_id

,sqlserver.plan\_handle

,sqlserver.session\_id

,sqlserver.sql\_text

,sqlserver.query\_hash

,sqlserver.query\_plan\_hash

)

WHERE ([sqlserver].[is\_system]=0)

),

ADD EVENT sqlserver.exchange\_spill

(

ACTION

(

sqlserver.database\_id

,sqlserver.plan\_handle

,sqlserver.session\_id

,sqlserver.sql\_text

,sqlserver.query\_hash

,sqlserver.query\_plan\_hash

)

WHERE ([sqlserver].[is\_system]=0)

)

ADD TARGET package0.ring\_buffer;

GO

-- Analyze the results

DROP TABLE IF EXISTS #tmpXML;

CREATE TABLE #tmpXML

(

EventTime DATETIME2(7) NOT NULL,

[Event] XML

);

DECLARE

@TargetData XML;

SELECT

@TargetData = CONVERT(XML,st.target\_data)

FROM

sys.dm\_xe\_sessions s WITH (NOLOCK)

JOIN sys.dm\_xe\_session\_targets st WITH(NOLOCK) ON

s.address = st.event\_session\_address

WHERE

s.name = 'Spills' and st.target\_name = 'ring\_buffer';

INSERT INTO #tmpXML(EventTime, [Event])

SELECT

t.e.value('@timestamp','datetime'), t.e.query('.')

FROM

@TargetData.nodes('/RingBufferTarget/event') AS t(e);

;WITH EventInfo

AS

(

SELECT

t.EventTime

,t.[Event].value('/event[1]/@name','sysname') AS [Event]

,t.

[Event].value('(/event[1]/action[@name="session\_id"]/value/text()

)[1]'

,'smallint') AS [Session]

,t.

[Event].value('(/event[1]/action[@name="database\_id"]/value/text(

))[1]'

,'smallint') AS [DB]

,t.

[Event].value('(/event[1]/action[@name="sql\_text"]/value/text())

[1]'

,'nvarchar(max)') AS [SQL]

,t.[Event]

.value('(/event[1]/data[@name="granted\_memory\_kb"]/value/text())

[1]'

,'bigint') AS [Granted Memory (KB)]

,t.[Event]

.value('(/event[1]/data[@name="used\_memory\_kb"]/value/text())[1]'

,'bigint') AS [Used Memory (KB)]

,t.[Event]

.value('xs:hexBinary((/event[1]/action[@name="plan\_handle"]/value

/text())[1])'

,'varbinary(64)') AS [PlanHandle]

,t.

[Event].value('(/event[1]/action[@name="query\_hash"]/value/text()

)[1]'

,'nvarchar(64)') AS [QueryHash]

,t.[Event]

.value('(/event[1]/action[@name="query\_plan\_hash"]/value/text())

[1]'

,'nvarchar(64)') AS [QueryPlanHash]

FROM

#tmpXML t

)

SELECT

ei.\*, qp.query\_plan

FROM

EventInfo ei

OUTER APPLY sys.dm\_exec\_query\_plan(ei.PlanHandle) qp

OPTION (RECOMPILE, MAXDOP 1);

Listing 9-11. Counting the number of spills

CREATE EVENT SESSION [Spill\_Count]

ON SERVER

ADD EVENT sqlserver.exchange\_spill,

ADD EVENT sqlserver.hash\_warning,

ADD EVENT sqlserver.sort\_warning

ADD TARGET package0.event\_counter;

DECLARE

@TargetData XML

SELECT

@TargetData = CONVERT(XML,st.target\_data)

FROM

sys.dm\_xe\_sessions s WITH (NOLOCK)

JOIN sys.dm\_xe\_session\_targets st WITH(NOLOCK) ON

s.address = st.event\_session\_address

WHERE

s.name = 'Spill Count' and st.target\_name = 'event\_counter';

;WITH EventInfo

AS

(

SELECT

t.e.value('@name','sysname') AS [Event]

,t.e.value('@count','bigint') AS [Count]

FROM

@TargetData.nodes

('/CounterTarget/Packages/Package[@name="sqlserver"]/Event')

AS t(e)

)

SELECT [Event], [Count]

FROM EventInfo

OPTION (RECOMPILE, MAXDOP 1);

Listing 9-12, which captures current PAGELATCH waits using the

sys.dm\_os\_waiting\_tasks view.

Listing 9-12. Capturing currently waiting sessions

-- SQL Server 2005-2017

SELECT

wt.session\_id

,wt.wait\_type

,er.wait\_resource

,er.wait\_time

FROM

sys.dm\_os\_waiting\_tasks wt WITH (NOLOCK)

JOIN sys.dm\_exec\_requests er WITH (NOLOCK) ON

wt.session\_id = er.session\_id

WHERE

wt.wait\_type LIKE 'PAGELATCH%'

OPTION (MAXDOP 1, RECOMPILE);

-- SQL Server 2019

SELECT

wt.session\_id

,wt.wait\_type

,er.wait\_resource

,er.wait\_time

,pi.database\_id

,pi.file\_id

,pi.page\_id

,pi.object\_id

,OBJECT\_NAME(pi.object\_id,pi.database\_id) as [object]

,pi.index\_id

,pi.page\_type\_desc

FROM

sys.dm\_os\_waiting\_tasks wt WITH (NOLOCK)

JOIN sys.dm\_exec\_requests er WITH (NOLOCK) ON

wt.session\_id = er.session\_id

CROSS APPLY

sys.fn\_PageResCracker(er.page\_resource) pc

CROSS APPLY

sys.dm\_db\_page\_info(pc.db\_id,pc.file\_id

,pc.page\_id,'DETAILED') pi

WHERE

wt.wait\_type LIKE 'PAGELATCH%'

OPTION (MAXDOP 1, RECOMPILE);

Listing 9-13. Capturing latch waits

CREATE EVENT SESSION [Latch Waits] ON SERVER

ADD EVENT sqlserver.latch\_suspend\_end

ADD TARGET package0.ring\_buffer

(SET max\_events\_limit=2000);

GO

-- The code below parses collected results

DROP TABLE IF EXISTS #tmpXML;

CREATE TABLE #tmpXML

(

EventTime DATETIME2(7) NOT NULL,

[Event] XML

);

DECLARE

@TargetData XML;

SELECT

@TargetData = CONVERT(XML,st.target\_data)

FROM

sys.dm\_xe\_sessions s WITH (NOLOCK)

JOIN sys.dm\_xe\_session\_targets st WITH(NOLOCK) ON

s.address = st.event\_session\_address

WHERE

s.name = 'Latch Waits' and st.target\_name = 'ring\_buffer';

INSERT INTO #tmpXML(EventTime, [Event])

SELECT t.e.value('@timestamp','datetime'), t.e.query('.')

FROM @TargetData.nodes('/RingBufferTarget/event') AS t(e);

;WITH EventInfo

AS

(

SELECT

t.[EventTime] as [Time]

,t.

[Event].value('(/event[1]/data[@name="database\_id"]/value/text())

[1]'

,'smallint') AS [DB]

,t.

[Event].value('(/event[1]/data[@name="duration"]/value/text())

[1]'

,'bigint') AS [Duration]

FROM

#tmpXML t

)

SELECT

MONTH([Time]) as [Month]

,DAY([Time]) as [Day]

,DATEPART(hour,[Time]) as [Hour]

,DATEPART(minute,[Time]) as [Minute]

,[DB]

,COUNT(\*) as [Latch Count]

,CONVERT(DECIMAL(15,3),SUM(Duration / 1000.)) as [Duration

(MS)]

FROM

EventInfo ei

GROUP BY

MONTH([Time]),DAY([Time]),DATEPART(hour,

[Time]),DATEPART(minute,[Time]),[DB]

ORDER BY

[Month],[Day],[Hour],[Minute],[DB]

OPTION (RECOMPILE, MAXDOP 1);

Listing 9-14. Detecting the sessions that consume the most tempdb space

;WITH SpaceUsagePages

AS

(

SELECT

ss.session\_id

,ss.user\_objects\_alloc\_page\_count +

ISNULL(SUM(ts.user\_objects\_alloc\_page\_count),0)

AS [user\_alloc\_page\_count]

,ss.user\_objects\_dealloc\_page\_count +

ISNULL(SUM(ts.user\_objects\_dealloc\_page\_count),0)

AS [user\_dealloc\_page\_count]

,ss.user\_objects\_deferred\_dealloc\_page\_count

AS [user\_deferred\_page\_count]

,ss.internal\_objects\_alloc\_page\_count +

ISNULL(SUM(ts.internal\_objects\_alloc\_page\_count),0)

AS [internal\_alloc\_page\_count]

,ss.internal\_objects\_dealloc\_page\_count +

ISNULL(SUM(ts.internal\_objects\_dealloc\_page\_count),0)

AS [internal\_dealloc\_page\_count]

FROM

sys.dm\_db\_session\_space\_usage ss WITH (NOLOCK) LEFT JOIN

sys.dm\_db\_task\_space\_usage ts WITH (NOLOCK) ON

ss.session\_id = ts.session\_id

GROUP BY

ss.session\_id

,ss.user\_objects\_alloc\_page\_count

,ss.user\_objects\_dealloc\_page\_count

,ss.internal\_objects\_alloc\_page\_count

,ss.internal\_objects\_dealloc\_page\_count

,ss.user\_objects\_deferred\_dealloc\_page\_count

)

,SpaceUsage

AS

(

SELECT

session\_id

,CONVERT(DECIMAL(12,3),

([user\_alloc\_page\_count] - [user\_dealloc\_page\_count])

/ 128.

) AS [user\_used\_mb]

,CONVERT(DECIMAL(12,3),

([internal\_alloc\_page\_count] -

[internal\_dealloc\_page\_count]) / 128.

) AS [internal\_used\_mb]

,CONVERT(DECIMAL(12,3),user\_deferred\_page\_count / 128.)

AS [user\_deferred\_used\_mb]

FROM

SpaceUsagePages

)

SELECT

su.session\_id

,su.user\_used\_mb

,su.internal\_used\_mb

,su.user\_deferred\_used\_mb

,su.user\_used\_mb + su.internal\_used\_mb AS [space\_used\_mb]

,es.open\_transaction\_count

,es.login\_time

,es.original\_login\_name

,es.host\_name

,es.program\_name

,er.status as [request\_status]

,er.start\_time

,CONVERT(DECIMAL(21,3),er.total\_elapsed\_time / 1000.) AS

[duration]

,er.cpu\_time

,ib.event\_info as [buffer]

,er.wait\_type

,er.wait\_time

,er.wait\_resource

,er.blocking\_session\_id

FROM

SpaceUsage su

LEFT JOIN sys.dm\_exec\_requests er WITH (NOLOCK) ON

su.session\_id = er.session\_id

LEFT JOIN sys.dm\_exec\_sessions es WITH (NOLOCK) ON

su.session\_id = es.session\_id

OUTER APPLY

sys.dm\_exec\_input\_buffer(es.session\_id,

er.request\_id) ib

WHERE

su.user\_used\_mb + su.internal\_used\_mb >= 50

ORDER BY

[space\_used\_mb] DESC

OPTION (RECOMPILE)

# Chapter 9. Latches

Chapter 9. Latches

Latches are lightweight synchronization objects that protect the consistency of SQL Server internal data structures. As the opposite of locks, which protect transactional data consistency, latches prevent corruption of the data structures in memory. In most cases, latches are short-lived and may be unnoticeable in systems with light loads. However, as loads grow, latch contention may become an issue and can limit system scalability and throughput.

Let’s look at a hypothetical scenario in which you want to log application request information in the database. Below code creates a few tables to store the data.

**#Creating tables to store the logs**

CREATE TABLE dbo.WebRequests\_Disk

(

RequestId INT NOT NULL identity(1,1),

RequestTime DATETIME2(4) NOT NULL

CONSTRAINT DEF\_WebRequests\_Disk\_RequestTime

DEFAULT SYSUTCDATETIME(),

URL VARCHAR(255) NOT NULL,

RequestType TINYINT NOT NULL,

ClientIP VARCHAR(15) NOT NULL,

BytesReceived INT NOT NULL,

CONSTRAINT PK\_WebRequests\_Disk

PRIMARY KEY NONCLUSTERED(RequestID)

ON [LOGDATA]

);

CREATE UNIQUE CLUSTERED INDEX

IDX\_WebRequests\_Disk\_RequestTime\_RequestId

ON dbo.WebRequests\_Disk(RequestTime,RequestId)

ON [LOGDATA];

CREATE TABLE dbo.WebRequestHeaders\_Disk

(

RequestId INT NOT NULL,

HeaderName VARCHAR(64) NOT NULL,

HeaderValue VARCHAR(256) NOT NULL,

CONSTRAINT PK\_WebRequestHeaders\_Disk

PRIMARY KEY CLUSTERED(RequestID,HeaderName)

ON [LOGDATA]

);

CREATE TABLE dbo.WebRequestParams\_Disk

(

RequestId INT NOT NULL,

ParamName VARCHAR(64) NOT NULL,

ParamValue nVARCHAR(256) NOT NULL,

CONSTRAINT PK\_WebRequestParams\_Disk

PRIMARY KEY CLUSTERED(RequestID,ParamName)

ON [LOGDATA]

);

The code below detects indexes that contribute to hotspots. This is a very simplified implementation,

**#Analyzing page latch index statistics**

SELECT

s.name + '.' + t.name as [table]

,i.index\_id

,i.name as [index]

,SUM(os.page\_latch\_wait\_count) AS [latch count]

,SUM(os.page\_latch\_wait\_in\_ms) as [latch wait (ms)]

FROM

sys.indexes i WITH (NOLOCK) JOIN sys.tables t WITH (NOLOCK)

on

i.object\_id = t.object\_id

JOIN sys.schemas s WITH (NOLOCK) ON

t.schema\_id = s.schema\_id

CROSS APPLY

sys.dm\_db\_index\_operational\_stats

(

DB\_ID()

,t.object\_id

,i.index\_id

,0

) os

GROUP BY

s.name, t.name, i.name, i.index\_id

ORDER BY

SUM(os.page\_latch\_wait\_in\_ms) DESC;

**####**

**Enable OPTIMIZE\_FOR\_SEQUENTIAL\_KEY option (SQL**

**Server 2019 and above)**

ALTER INDEX PK\_WebRequestHeaders\_Disk

ON dbo.WebRequestHeaders\_Disk

SET (OPTIMIZE\_FOR\_SEQUENTIAL\_KEY = ON);

ALTER INDEX PK\_WebRequestParams\_Disk

ON dbo.WebRequestParams\_Disk

SET (OPTIMIZE\_FOR\_SEQUENTIAL\_KEY = ON);

Below code shows such an example. It redefines two of the tables from, partitioning them with a new HashVal column calculated as CHECKSUM(RequestId). This randomly distributes the data across multiple partitions, reducing insertion rates and latch contention on each individual partition. Note that the HashVal column is defined as the rightmost column in the indexes, to preserve the sorting order on each individual partition.

**#Implementing hash partitioning**

-- For demo purposes

TRUNCATE TABLE dbo.WebRequests\_Disk;

DROP TABLE dbo.WebRequestHeaders\_Disk;

DROP TABLE dbo.WebRequestParams\_Disk;

GO

CREATE PARTITION FUNCTION pfHash(int)

AS RANGE LEFT FOR VALUES

(-1847483647,-1547483647,-1247483647,-947483647,-647483647,-34748

3647

,-47483647,252516353,552516353,852516353,1152516353,1452516353,17

52516353);

CREATE PARTITION SCHEME psHash

AS PARTITION pfHash

ALL TO ([LOGDATA]);

CREATE TABLE dbo.WebRequestHeaders\_Disk

(

RequestId INT NOT NULL,

HeaderName VARCHAR(64) NOT NULL,

HeaderValue VARCHAR(256) NOT NULL,

HashVal AS CHECKSUM(RequestId) PERSISTED,

CONSTRAINT PK\_WebRequestHeaders\_Disk

PRIMARY KEY CLUSTERED(RequestID,HeaderName,HashVal)

ON psHash(HashVal)

);

CREATE TABLE dbo.WebRequestParams\_Disk

(

RequestId INT NOT NULL,

ParamName VARCHAR(64) NOT NULL,

ParamValue nVARCHAR(256) NOT NULL,

HashVal AS CHECKSUM(RequestId) PERSISTED,

CONSTRAINT PK\_WebRequestParams\_Disk

PRIMARY KEY CLUSTERED(RequestID,ParamName,HashVal)

ON psHash(HashVal)

);

you can get information about individual latch types using the sys.dm\_os\_latch\_stats view shown below. You can also clear latch statistics with the command DBCC SQLPERF(’sys.dm\_os\_latch\_stats', CLEAR).

**#Analyzing latch statistics**

;WITH Latches

AS

(

SELECT

latch\_class, wait\_time\_ms, waiting\_requests\_count

,100. \* wait\_time\_ms / SUM(wait\_time\_ms) OVER() AS Pct

,100. \* SUM(wait\_time\_ms) OVER(ORDER BY wait\_time\_ms

DESC) /

NULLIF(SUM(wait\_time\_ms) OVER(), 0) AS RunningPct

,ROW\_NUMBER() OVER(ORDER BY wait\_time\_ms DESC) AS RowNum

FROM

sys.dm\_os\_latch\_stats WITH (NOLOCK)

WHERE

wait\_time\_ms > 0 AND

latch\_class NOT IN (N'BUFFER',N'SLEEP\_TASK')

)

SELECT

l1.latch\_class AS [Latch Type]

,l1.waiting\_requests\_count AS [Latch Count]

,CONVERT(DECIMAL(12,3), l1.wait\_time\_ms / 1000.0)

AS [Wait Time]

,CONVERT(DECIMAL(12,1), l1.wait\_time\_ms /

l1.waiting\_requests\_count)

AS [Avg Wait Time]

,CONVERT(DECIMAL(6,3), l1.Pct)

AS [Percent]

,CONVERT(DECIMAL(6,3), l1.RunningPct)

AS [Running Percent]

FROM

Latches l1

WHERE

l1.RunningPct <= 99 OR l1.RowNum = 1

ORDER BY

l1.RunningPct

OPTION (RECOMPILE, MAXDOP 1);

# Chapter 10. Transaction Log

The code below shows the overhead involved in autocommitted transactions as compared to explicit transactions. It performs an INSERT/UPDATE/DELETE sequence 10,000 times in the loop, in autocommitted and explicit transactions, respectively. It then compares their execution time and transaction log throughput using the sys.dm\_io\_virtual\_file\_stats view.

**Explicit and Autocommitted Transactions and Log Overhead**

**#Explicit and autocommitted transactions**

CREATE TABLE dbo.TranOverhead

(

Id INT NOT NULL,

Col CHAR(50) NULL,

CONSTRAINT PK\_TranOverhead

PRIMARY KEY CLUSTERED(Id)

);

-- Autocommitted transactions

DECLARE

@Id INT = 1

,@StartTime DATETIME = GETDATE()

,@num\_of\_writes BIGINT

,@num\_of\_bytes\_written BIGINT

SELECT @num\_of\_writes = num\_of\_writes, @num\_of\_bytes\_written =

num\_of\_bytes\_written

FROM sys.dm\_io\_virtual\_file\_stats(db\_id(),2);

WHILE @Id <= 10000

BEGIN

INSERT INTO dbo.TranOverhead(Id, Col) VALUES(@Id, 'A');

UPDATE dbo.TranOverhead SET Col = 'B' WHERE Id = @Id;

DELETE FROM dbo.TranOverhead WHERE Id = @Id;

SET @Id += 1;

END;

SELECT

DATEDIFF(MILLISECOND,@StartTime,GETDATE())

AS [Time(ms): Autocommitted Tran]

,s.num\_of\_writes - @num\_of\_writes

AS [Number of writes]

,(s.num\_of\_bytes\_written - @num\_of\_bytes\_written) / 1024

AS [Bytes written (KB)]

FROM

sys.dm\_io\_virtual\_file\_stats(db\_id(),2) s;

GO

-- Explicit Tran

DECLARE

@Id INT = 1

,@StartTime DATETIME = GETDATE()

,@num\_of\_writes BIGINT

,@num\_of\_bytes\_written BIGINT

SELECT @num\_of\_writes = num\_of\_writes, @num\_of\_bytes\_written =

num\_of\_bytes\_written

FROM sys.dm\_io\_virtual\_file\_stats(db\_id(),2);

WHILE @Id <= 10000

BEGIN

BEGIN TRAN

INSERT INTO dbo.TranOverhead(Id, Col) VALUES(@Id, 'A');

UPDATE dbo.TranOverhead SET Col = 'B' WHERE Id = @Id;

DELETE FROM dbo.TranOverhead WHERE Id = @Id;

COMMIT

SET @Id += 1;

END;

SELECT

DATEDIFF(MILLISECOND,@StartTime,GETDATE())

AS [Time(ms): Explicit Tran]

,s.num\_of\_writes - @num\_of\_writes

AS [Number of writes]

,(s.num\_of\_bytes\_written - @num\_of\_bytes\_written) / 1024

AS [Bytes written (KB)]

FROM

sys.dm\_io\_virtual\_file\_stats(db\_id(),2) s;

**Delayed Durability**

**Below code shows how to specify it at the transaction level.**

**#Controlling delayed durability on transaction level**

BEGIN TRAN

/\* Do the work \*/

COMMIT WITH (DELAYED\_DURABILITY=ON);

Delayed durability may be used in chatty systems with large numbers of autocommitted transactions and insufficient log throughput. In most cases, avoid it. use it as a last resort, only when all other log throughput improvement techniques have been unsuccessful and only when data loss is acceptable.

**VIRTUAL LOG FILES**

Internally, SQL Server divides physical log files into smaller parts called Virtual Log Files (VLF). SQL Server uses them as a unit of management, and they can be active or inactive.

Active VLFs store the active portion of the transaction log, which contains log records required to keep the database transactionally consistent, provide point-in-time recovery, and support active SQL Server processes such as transactional replication and AlwaysOn Availability Groups. An inactive VLF contains the truncated (inactive) and unused parts of the transaction log.

**#Analyzing VLFs in the database**

SELECT \*

FROM sys.dm\_db\_log\_info(DB\_ID());

SELECT

COUNT(\*) as [VLF Count]

,MIN(vlf\_size\_mb) as [Min VLF Size (MB)]

,MAX(vlf\_size\_mb) as [Max VLF Size (MB)]

,AVG(vlf\_size\_mb) as [Avg VLF Size (MB)]

FROM sys.dm\_db\_log\_info(DB\_ID());

analyze VLFs by using the sys.dm\_db\_log\_info data management view in SQL Server 2016 and above or with the DBCC LOGINFO command in older versions of SQL Server. shows the code that uses this view against one of the databases.

**Log Truncation Issues**

If you end up in a Transaction Log Full situation, my first and most important advice is: Don’t panic. First, you need to analyze the root cause of the issue and see if you can mitigate it quickly. You can do this by looking at the log\_reuse\_wait\_desc column in the sys.databases view, either querying it directly or using the more sophisticated version shown in below code. This column shows you why the log is not truncated.

**#Analyzing the log\_reuse\_wait\_desc column in the**

**sys.databases view**

CREATE TABLE #SpaceUsed

(

database\_id SMALLINT NOT NULL,

file\_id SMALLINT NOT NULL,

space\_used DECIMAL(15,3) NOT NULL,

PRIMARY KEY(database\_id, file\_id)

);

EXEC master..sp\_MSforeachdb

N'USE[?];

INSERT INTO #SpaceUsed(database\_id, file\_id, space\_used)

SELECT DB\_ID(''?''), file\_id,

(size - CONVERT(INT,FILEPROPERTY(name, ''SpaceUsed''))) /

128.

FROM sys.database\_files

WHERE type = 1;';

SELECT

d.database\_id, d.name, d.recovery\_model\_desc

,d.state\_desc, d.log\_reuse\_wait\_desc, m.physical\_name

,m.is\_percent\_growth

,IIF(m.is\_percent\_growth = 1

,m.growth

,CONVERT(DECIMAL(15,3),m.growth / 128.0)

) AS [Growth (MB or %)]

,CONVERT(DECIMAL(15,3),m.size / 128.0) AS [Size (MB)]

,IIF(m.max\_size = -1

,-1

,CONVERT(DECIMAL(15,3),m.max\_size / 128.0)

) AS [Max Size(MB)]

,s.space\_used as [Space Used(MB)]

FROM

sys.databases d WITH (NOLOCK)

JOIN sys.master\_files m WITH (NOLOCK) ON

d.database\_id = m.database\_id

LEFT OUTER JOIN #SpaceUsed s ON

s.database\_id = m.database\_id AND

s.file\_id = m.file\_id

ORDER BY

d.database\_id;

**#Getting active transactions**

You can see the list of active transactions using the code from code below. The code may provide you multiple rows for each transaction, because it gets log usage information on a per-database basis.

SELECT

dt.database\_id

,DB\_NAME(dt.database\_id) as [DB]

,st.session\_id

,CASE at.transaction\_state

WHEN 0 THEN 'Not Initialized'

WHEN 1 THEN 'Not Started'

WHEN 2 THEN 'Active'

WHEN 3 THEN 'Ended (R/O)'

WHEN 4 THEN 'Commit Initialize'

WHEN 5 THEN 'Prepared'

WHEN 6 THEN 'Committed'

WHEN 7 THEN 'Rolling Back'

WHEN 8 THEN 'Rolled Back'

END AS [State]

,at.transaction\_begin\_time

,es.login\_name

,ec.client\_net\_address

,ec.connect\_time

,dt.database\_transaction\_log\_bytes\_used

,dt.database\_transaction\_log\_bytes\_reserved

,er.status

,er.wait\_type

,er.last\_wait\_type

,sql.text AS [SQL]

FROM

sys.dm\_tran\_database\_transactions dt WITH (NOLOCK)

JOIN sys.dm\_tran\_session\_transactions st WITH (NOLOCK) ON

dt.transaction\_id = st.transaction\_id

JOIN sys.dm\_tran\_active\_transactions at WITH (NOLOCK) ON

dt.transaction\_id = at.transaction\_id

JOIN sys.dm\_exec\_sessions es WITH (NOLOCK) ON

st.session\_id = es.session\_id

JOIN sys.dm\_exec\_connections ec WITH (NOLOCK) ON

st.session\_id = ec.session\_id

LEFT OUTER JOIN sys.dm\_exec\_requests er WITH (NOLOCK) ON

st.session\_id = er.session\_id

CROSS APPLY

sys.dm\_exec\_sql\_text(ec.most\_recent\_sql\_handle) sql

ORDER BY

dt.database\_transaction\_begin\_timel

You can kill the session that holds active transaction using the KILL command. Later, you can analyze why the transaction was not properly managed.

The code shows you can use to monitor the health of Availability Groups. You need to run it on the primary node to get the right results.

**#Availability Group monitoring code**

SELECT

ar.replica\_server\_name as [Replica]

,DB\_NAME(drs.database\_id) AS DB

,drs.synchronization\_state\_desc as [Sync State]

,ars.synchronization\_health\_desc as [Health]

,ar.availability\_mode as [Syncronous]

,drs.log\_send\_queue\_size

,drs.redo\_queue\_size

,ISNULL(

GhostReplicaState.max\_low\_water\_mark\_for\_ghosts -

drs.low\_water\_mark\_for\_ghosts,0

) AS [water\_mark\_diff]

,drs.log\_send\_rate

,drs.redo\_rate

,pri.last\_commit\_time AS primary\_last\_commit\_time

,IIF(drs.is\_primary\_replica = 1

,pri.last\_commit\_time

,drs.last\_commit\_time

) AS node\_last\_commit\_time

,IIF(drs.is\_primary\_replica = 1

,0

,DATEDIFF(ms,drs.last\_commit\_time,pri.last\_commit\_time)

) AS commit\_latency

FROM

sys.availability\_groups ag WITH (NOLOCK)

JOIN sys.availability\_replicas ar WITH (NOLOCK) ON

ag.group\_id = ar.group\_id

JOIN sys.dm\_hadr\_availability\_replica\_states ars WITH

(NOLOCK) ON

ar.replica\_id = ars.replica\_id

JOIN sys.dm\_hadr\_database\_replica\_states drs WITH (NOLOCK)

ON

ag.group\_id = drs.group\_id AND

drs.replica\_id = ars.replica\_id

LEFT JOIN sys.dm\_hadr\_database\_replica\_states pri WITH

(NOLOCK) ON

pri.is\_primary\_replica = 1 AND

drs.database\_id = pri.database\_id

OUTER APPLY

(

SELECT MAX(drs2.low\_water\_mark\_for\_ghosts) AS

max\_low\_water\_mark\_for\_ghosts

FROM sys.dm\_hadr\_database\_replica\_states drs2 WITH

(NOLOCK)

WHERE drs.database\_id = drs2.database\_id

) GhostReplicaState

WHERE

ars.is\_local = 0

ORDER BY

replica\_server\_name, DB;

**#Creating xEvent sessions for Availability Group performance**

**troubleshooting.**

-- Create on primary node

CREATE EVENT SESSION [AlwaysOn\_Tracing\_Primary] ON SERVER

ADD EVENT sqlserver.hadr\_capture\_log\_block,

ADD EVENT sqlserver.hadr\_db\_commit\_mgr\_harden,

ADD EVENT sqlserver.hadr\_db\_commit\_mgr\_harden\_still\_waiting,

ADD EVENT sqlserver.hadr\_log\_block\_compression,

ADD EVENT sqlserver.hadr\_log\_block\_send\_complete,

ADD EVENT sqlserver.hadr\_receive\_harden\_lsn\_message,

ADD EVENT sqlserver.log\_flush\_complete,

ADD EVENT sqlserver.log\_flush\_start

ADD TARGET package0.ring\_buffer(SET max\_events\_limit=

(0),max\_memory=(16384));

GO

-- Create on secondary node

CREATE EVENT SESSION [AlwaysOn\_Tracing\_Secondary] ON SERVER

ADD EVENT sqlserver.hadr\_apply\_log\_block,

ADD EVENT sqlserver.hadr\_log\_block\_decompression,

ADD EVENT sqlserver.hadr\_lsn\_send\_complete,

ADD EVENT sqlserver.hadr\_send\_harden\_lsn\_message,

ADD EVENT sqlserver.hadr\_transport\_receive\_log\_block\_message,

ADD EVENT sqlserver.log\_flush\_complete,

ADD EVENT sqlserver.log\_flush\_start

ADD TARGET package0.ring\_buffer(SET max\_events\_limit=

(0),max\_memory=(16384));

shows two xEvent sessions you can run to capture xEvents. word of caution: those sessions can collect a large number of events. Do not keep them running outside of the troubleshooting.

You can use the log\_block\_id and database\_id fields in both sessions to correlate session data. There is a catch, though: in hadr\_send\_harden\_lsn\_message, hadr\_receive\_harden\_lsn\_message, and hadr\_lsn\_send\_complete events, the log\_block\_id will be higher than in previous events. This is due to how xEvents collects the data. The difference in values depends on the load in the database; however, it won’t exceed 120.

# Chapter 10 : Always On Availability Groups

Listing 12-1. Availability Group monitoring code

SELECT

ar.replica\_server\_name as [Replica]

,DB\_NAME(drs.database\_id) AS DB

,drs.synchronization\_state\_desc as [Sync State]

,ars.synchronization\_health\_desc as [Health]

,ar.availability\_mode as [Syncronous]

,drs.log\_send\_queue\_size

,drs.redo\_queue\_size

,ISNULL(

GhostReplicaState.max\_low\_water\_mark\_for\_ghosts -

drs.low\_water\_mark\_for\_ghosts,0

) AS [water\_mark\_diff]

,drs.log\_send\_rate

,drs.redo\_rate

,pri.last\_commit\_time AS primary\_last\_commit\_time

,IIF(drs.is\_primary\_replica = 1

,pri.last\_commit\_time

,drs.last\_commit\_time

) AS node\_last\_commit\_time

,IIF(drs.is\_primary\_replica = 1

,0

,DATEDIFF(ms,drs.last\_commit\_time,pri.last\_commit\_time)

) AS commit\_latency

FROM

sys.availability\_groups ag WITH (NOLOCK)

JOIN sys.availability\_replicas ar WITH (NOLOCK) ON

ag.group\_id = ar.group\_id

JOIN sys.dm\_hadr\_availability\_replica\_states ars WITH

(NOLOCK) ON

ar.replica\_id = ars.replica\_id

JOIN sys.dm\_hadr\_database\_replica\_states drs WITH (NOLOCK)

ON

ag.group\_id = drs.group\_id AND

drs.replica\_id = ars.replica\_id

LEFT JOIN sys.dm\_hadr\_database\_replica\_states pri WITH

(NOLOCK) ON

pri.is\_primary\_replica = 1 AND

drs.database\_id = pri.database\_id

OUTER APPLY

(

SELECT MAX(drs2.low\_water\_mark\_for\_ghosts) AS

max\_low\_water\_mark\_for\_ghosts

FROM sys.dm\_hadr\_database\_replica\_states drs2 WITH

(NOLOCK)

WHERE drs.database\_id = drs2.database\_id

) GhostReplicaState

WHERE

ars.is\_local = 0

ORDER BY

replica\_server\_name, DB;

Listing 12-2. Creating xEvent sessions for Availability Group performance

troubleshooting.

-- Create on primary node

CREATE EVENT SESSION [AlwaysOn\_Tracing\_Primary] ON SERVER

ADD EVENT sqlserver.hadr\_capture\_log\_block,

ADD EVENT sqlserver.hadr\_db\_commit\_mgr\_harden,

ADD EVENT sqlserver.hadr\_db\_commit\_mgr\_harden\_still\_waiting,

ADD EVENT sqlserver.hadr\_log\_block\_compression,

ADD EVENT sqlserver.hadr\_log\_block\_send\_complete,

ADD EVENT sqlserver.hadr\_receive\_harden\_lsn\_message,

ADD EVENT sqlserver.log\_flush\_complete,

ADD EVENT sqlserver.log\_flush\_start

ADD TARGET package0.ring\_buffer(SET max\_events\_limit=

(0),max\_memory=(16384));

GO

-- Create on secondary node

CREATE EVENT SESSION [AlwaysOn\_Tracing\_Secondary] ON SERVER

ADD EVENT sqlserver.hadr\_apply\_log\_block,

ADD EVENT sqlserver.hadr\_log\_block\_decompression,

ADD EVENT sqlserver.hadr\_lsn\_send\_complete,

ADD EVENT sqlserver.hadr\_send\_harden\_lsn\_message,

ADD EVENT sqlserver.hadr\_transport\_receive\_log\_block\_message,

ADD EVENT sqlserver.log\_flush\_complete,

ADD EVENT sqlserver.log\_flush\_start

ADD TARGET package0.ring\_buffer(SET max\_events\_limit=

(0),max\_memory=(16384));

Listing 12-3. Readable secondaries: Table creation

CREATE TABLE dbo.T1

( ID INT NOT NULL,

Placeholder CHAR(8000) NULL,

CONSTRAINT PK\_T1 PRIMARY KEY CLUSTERED(ID)

);

CREATE TABLE dbo.T2

(

Col INT

);

;WITH N1(C) AS (SELECT 0 UNION ALL SELECT 0) -- 2 rows

,N2(C) AS (SELECT 0 FROM N1 AS T1 CROSS JOIN N1 AS T2) -- 4 rows

,N3(C) AS (SELECT 0 FROM N2 AS T1 CROSS JOIN N2 AS T2) -- 16 rows

,N4(C) AS (SELECT 0 FROM N3 AS T1 CROSS JOIN N3 AS T2) -- 256 rows

,N5(C) AS (SELECT 0 FROM N4 AS T1 CROSS JOIN N4 AS T2 ) -- 65,536

rows

,IDS(ID) AS (SELECT ROW\_NUMBER() OVER (ORDER BY (SELECT NULL))

FROM N5)

INSERT INTO dbo.T1(ID)

SELECT ID FROM IDS;