
# Module 1: Architecture

*Student Lab Manual*

SQL Server 2012: Performance Tuning – Design, Internals, and Architecture

Version 1.0

**Conditions and Terms of Use**

**Microsoft Confidential - For Internal Use Only**

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.



**Copyright and Trademarks**



Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at
*http*://www.microsoft.com/about/legal/permissions/

Microsoft®, Internet Explorer®, and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

## Introduction

This Lab explores SQL Server 2012 SQLOS architecture, memory architecture as well as the Waits and Queue methodology.

## Objectives

After completing this lab, you will be able to:

- Learn the important monitoring DMVs
- Understand SQL Server memory management
- Understand the Waits and Queues methodology.

## Prerequisites

Before starting this lab verify the environment for the following

Make sure you have the appropriate Virtual machine started.

## Estimated time to complete this lab

90 minutes

# Exercise 1: Explore the SQLOS Execution, scheduler and thread DMV's

## Objectives

In this exercise, you will:

- Learn the most common SQLOS and Execution DMV's.

## Prerequisites

- Connect to the SQL2012PT Virtual Machine
- Log in using the following credentials
  *User:* Administrator
  *Password:* P@ssw0rd

> **Note:** The Virtual Machine for this workshop is time bombed for security purposes. You may need to rearm the virtual machine if the activation has expired. If the VM issues a message that it needs to be reactivated, you can use slmgr.vbs with the rearm option as follows:
>
> 1. Open an elevated command prompt (right click on "Command Prompt" in the Start menu and click "Run As Administrator)
> 2. Execute the following command
>    *slmgr.vbs –rearm*

## Task: Use DMVs and other methods to view sessions and tasks on the server

1. Establish a connection to the server from Management Studio
2. Copy and Paste the following query in a New Query window (you can also find the code in the ViewSessionsDMVs.sql script in C:\Labs\Module1\LabFiles\Exercise1). Explore the 3 DMVs below by executing each query individually, we will be using these DMV's in more complex examples shortly

```
-- This DMV returns one row per scheduler. Schedulers that are marked as
VISIBLE_ONLINE service user requests.
-- Also important as it lists number of workers, active tasks, queued
tasks, and the Active_worker_address
-- There are other types of schedulers including ones for backups, DAC etc.
-- More details: http://msdn.microsoft.com/en-us/library/ms177526.aspx

select * from sys.dm_os_schedulers

--Returns information about each request that is executing within SQL
Server
select * from sys.dm_exec_requests
```

```
-- Returns one row per authenticated session on SQL Server.
sys.dm_exec_sessions is a server-scope view that shows information about
all active user connections and internal tasks. This information includes
client version, client program name, client login time, login user, current
session setting
select * from sys.dm_exec_sessions
```

3. Open a Command Prompt and run the following batch file which simulates a workload. This will give us some baseline activity to further explore the DMVs and Activity monitor.

   /Labs/Module1/LabFiles/Exercise1/BaselineWorkload/StartWorkLoad.cmd 5

   **Note:** the Second parameter is the number of connections – you can use more than 5 threads or less depending on performance. Also this assumes a default instance, if that is not the case add the Server\Instance as the second parameter

   Ex: /Labs/Module1/ LabFiles/Exercise1/BaselineWorkload/StartWorkLoad.cmd 5 Server\Instance

4. Let us explore getting a list of every "Running" request and what it is executing. The most simple form of the query is using a cross apply with sys.dm_exec_sql_text. Copy this query into Management Studio and run it to view the current requests (this is also found in the ViewSessionsDMVs.sql script). You may need to run this a few times before you catch any queries executing.

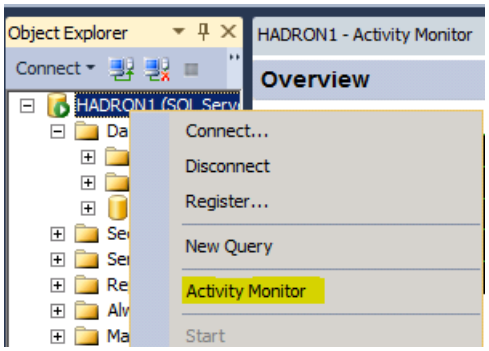
```
select session_id , S2.text
from sys.dm_exec_requests
CROSS APPLY sys.dm_exec_sql_text(sql_handle) as S2
```

   A more useful query exposing a few key columns is found below. Note that this covers only currently executing requests. If a session is idle, it won't appear in this resultset. Also the substring portion of this query is able to use the start and end offsets of a batch and give you the exact statement within a multi-line batch that is currently being executed. You will see the difference in the text and the sql_statement columns of the results. Copy this query into Management Studio and run it (found in the ViewSessionsDMVs.sql script). Again, you may need to run it a few times to catch the queries executing.

```
--- Looking at all currently Active Requests and the statements
that are running
select
a.session_id,
start_time,
b.host_name,
b.program_name,
DB_NAME(a.database_id) as DatabaseName,
a.status,
blocking_session_id,
wait_type,
wait_time,
wait_resource,
a.cpu_time,
a.total_elapsed_time,
```

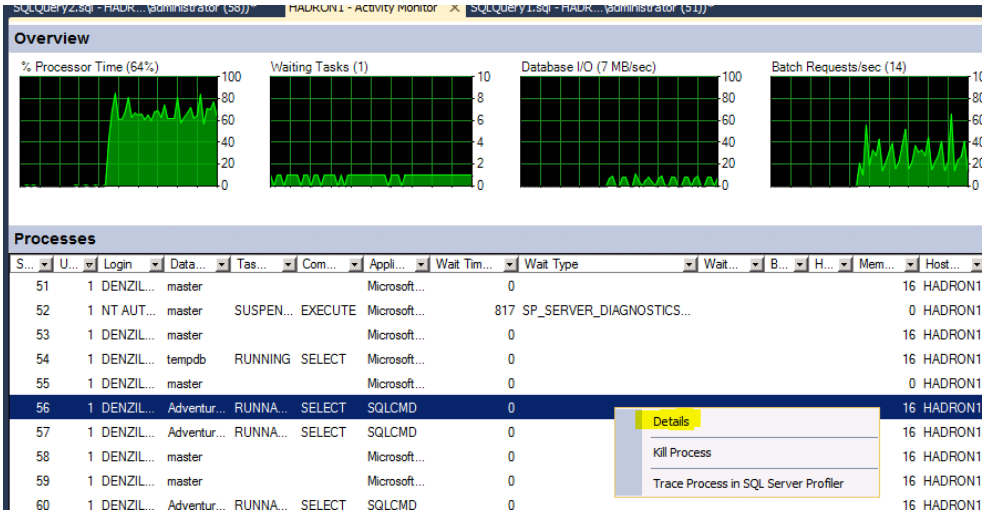
```
scheduler_id,
a.reads,
a.writes,
 (SELECT TOP 1 SUBSTRING(s2.text,statement_start_offset / 2+1 ,
      ( (CASE WHEN statement_end_offset = -1
         THEN (LEN(CONVERT(nvarchar(max),s2.text)) * 2)
         ELSE statement_end_offset END)  -
statement_start_offset) / 2+1))  AS sql_statement

, s2.text
from
sys.dm_exec_requests  a inner join
sys.dm_exec_sessions b on a.session_id = b.session_id
CROSS APPLY sys.dm_exec_sql_text(a.sql_handle) AS s2
```

5. Check the the same running queries now through the Activity Monitor. You can launch the Activity Monitor by right-clicking the server name in Management Studio as below:




Click the "Processes" bar to expand that section. In this view, you will see all the sessions.

- Click on the "Task State" column and choose "non-blanks" from the dropdown to see the currently executing queries.
- You can right click on any of the rows and click "Details" to see the query text
- Each of the columns have filters whereby you can filter the rows in this view.



6. As discussed, at any given point only one task is currently "Running" on the scheduler, the others being in the runnable queue or the wait queue. The schedulers DMV does have the active task's address which helps in finding out what exactly is currently running on the schedulers. The following query is also found in the ViewSessionsDMVs.sql script:

```
-- This also gives the Queries that are currently "Running" on
each Scheduler
-- We can get all kinds of other details from the thread ID to
resources it holds etc.
-- Note: on a single CPU box, this will show the query itself as
the only output.

select
a.scheduler_id ,
b.session_id,
 (SELECT TOP 1 SUBSTRING(s2.text,statement_start_offset / 2+1 ,
      ( (CASE WHEN statement_end_offset = -1
         THEN (LEN(CONVERT(nvarchar(max),s2.text)) * 2)
         ELSE statement_end_offset END)  -
statement_start_offset) / 2+1))  AS sql_statement
from
sys.dm_os_schedulers a inner join sys.dm_os_tasks b on
a.active_worker_address = b.worker_address
inner join sys.dm_exec_requests c on b.task_address =
c.task_address
CROSS APPLY sys.dm_exec_sql_text(c.sql_handle) AS s2
```

7. Once you have completed reviewing the queries, stop the batch script by typing "Enter" in the Command Prompt. Close Activity Monitor and any other queries you have open in Management Studio.

> **Note:** It is important to hit enter on the scenarios once they are done to run the cleanup batch file that cleans things up and resets them to the default values.

## Task: Use DMV's to identify the cause of a hang due to thread exhaustion

1. Set up the next task by running the following script from the Command Prompt:

   /Labs/Module1/LabFiles/Exercise1/ServerHang/Scenario.cmd

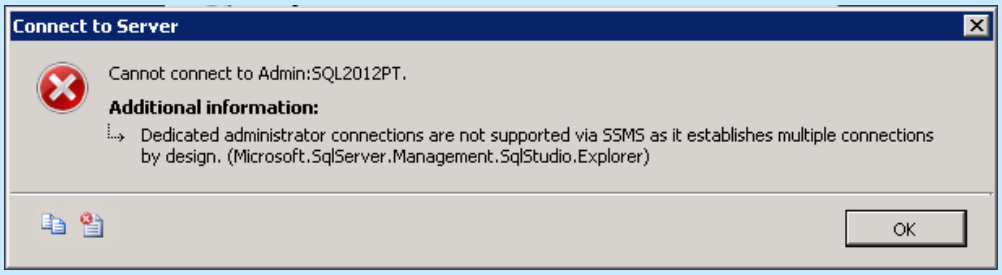
2. Give the script about 60 seconds or so. Once you see the line "Press ENTER to end the scenario", try to make a new connection to SQL Server by opening a new query

window in Management Studio as follows:
File → New → Database Engine Query.

The connection should fail as the server is hung at this point

3. Connect to the server using the Dedicated Admin connection by prefixing the server name with the word ADMIN, for example admin:ServerName

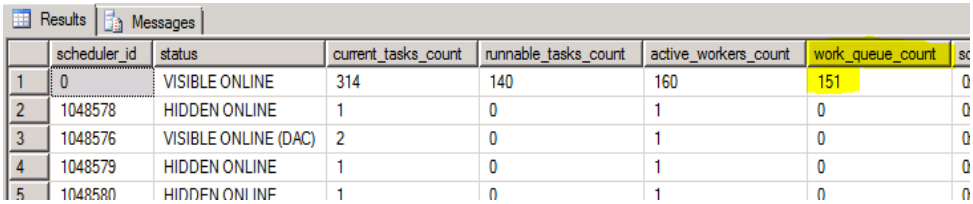

**Note:** If you get the following error message:




it indicates that you have attempted to connect to the Object Explorer with the dedicated administrator connection. Click OK, then click Cancel on the connection dialog box. In the SSMS menu, click File -> New -> Database Engine Query and then connect with the dedicated administrator connection as shown above. This should now connect successfully.

4. Run the following query (found in the ThreadExhaustion.sql script in C:\Labs\Module1\LabFiles\Exercise1) and you will see that the work_queue_count is greater than 0 which means that there are tasks that don't have an available worker thread to process them. This happens when existing workers are fully consumed in which case you won't be able to connect.

```
select scheduler_id,status,current_tasks_count,
runnable_tasks_count,active_workers_count,work_queue_count
from sys.dm_os_schedulers
```



Results | Messages

| | scheduler_id | status | current_tasks_count | runnable_tasks_count | active_workers_count | work_queue_count | sc |
|---|---|---|---|---|---|---|---|
| 1 | 0 | VISIBLE ONLINE | 314 | 140 | 160 | 151 | 0 |
| 2 | 1048578 | HIDDEN ONLINE | 1 | 0 | 1 | 0 | 0 |
| 3 | 1048576 | VISIBLE ONLINE (DAC) | 2 | 0 | 1 | 0 | 0 |
| 4 | 1048579 | HIDDEN ONLINE | 1 | 0 | 1 | 0 | 0 |
| 5 | 1048580 | HIDDEN ONLINE | 1 | 0 | 1 | 0 | 0 |

5. Now from the Dedicated Admin connection let's try to investigate why the workers are consumed by seeing what requests are running. You will see blocking occurring and several sessions waiting on a Lock (check the wait_type column).

```
select  session_id,start_time,status, blocking_session_id,
wait_type,wait_time, wait_resource, open_transaction_count
,s2.text
 from sys.dm_exec_requests a
 CROSS APPLY sys.dm_exec_sql_text(a.sql_handle) AS s2
where status <> 'background'
```

6. Let us try to find the head blocker now, and you will notice the head blocker is suspended and has an open transaction.

```
select
b.session_id,
start_time,
b.host_name,
b.program_name,
a.status,
b.open_transaction_count,
blocking_session_id,
wait_type,
wait_time,
wait_resource,
a.cpu_time,
a.Total_elapsed_time,
scheduler_id,
a.reads,
a.writes,
 (SELECT TOP 1 SUBSTRING(s2.text,statement_start_offset / 2+1
,
       ( (CASE WHEN statement_end_offset = -1
            THEN (LEN(CONVERT(nvarchar(max),s2.text)) * 2)
            ELSE statement_end_offset END)  -
statement_start_offset) / 2+1))  AS sql_statement
          , S2.text
from
sys.dm_exec_sessions b  left outer join
sys.dm_exec_requests  a  on a.session_id = b.session_id
CROSS APPLY sys.dm_exec_sql_text(a.sql_handle) AS s2
where b.session_id in (select blocking_session_id from
sys.dm_exec_requests z)
```

```
and (a.blocking_session_id  = 0 or blocking_session_id =
a.session_id)
```

7. Given the server is hung, let us Kill the head blocker to allow other connections through. Run the head blocker query again and you should see it empty. You may need to wait a few seconds for the blocking to clear before the head blocker query returns empty.

```
-- Replace the session_id with the value of the session_id you
got in the prior query
kill <session_id>
```

8. You should see that now the server will let connections through. Attempt to make a new connection to the server and you should be able to. Go to the command prompt running the scenario and hit enter. Close any open queries in Management Studio.

# Exercise 2: Waits and Queues Methodology

## Objectives

In this exercise, you will:

- Learn how to use the waits and queues methodology
- Learn how to identify the predominant wait type and the queries that contribute to that wait type.

## Task: Identifying the predominant wait type or bottleneck and the queries that contribute to that wait type.

1. Open a Command Prompt and run the following script to simulate a potential performance problem:

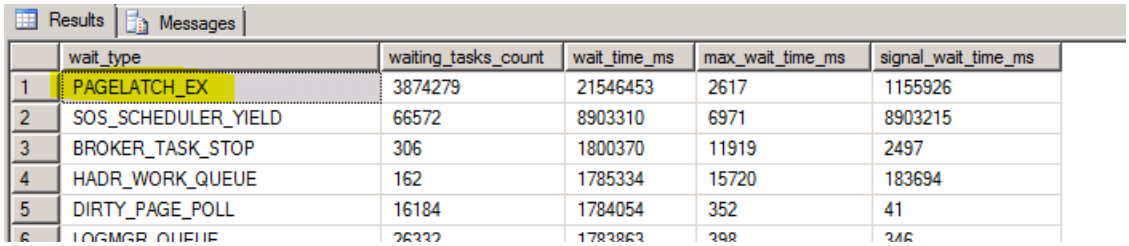
   /Labs/Module1/LabFiles/Exercise2/LatchWaits/Scenario.cmd

2. In order to find the bottleneck on our system, we are first going to look at cumulative waits. Whenever a session waits on a resource, the SQLOS records that wait. Examining the most common wait types on a system is the key to understanding which resources are causing a bottleneck in performance. In order to utilize and further develop this method of troubleshooting, a good understanding of the different wait types is necessary.

   First let us look at the sys.dm_os_wait_stats DMV. Note the waits represented here are since the SQL Server service was last restarted or since waitstats were explicitly reset manually. You can find the query below in the WaitStats.sql script in C:\Labs\Module1\LabFiles\Exercise2

```
-- Cumulative waits from server restart
-- Need to take snapshots and then calculate the Difference
select * from sys.dm_os_wait_stats
order by wait_time_ms desc
```

   We may or may not get our culprit here if we just look at a single snapshot individually as the waits could occur anytime since server restart.



Results | Messages

| | wait_type | waiting_tasks_count | wait_time_ms | max_wait_time_ms | signal_wait_time_ms |
|---|---|---|---|---|---|
| 1 | PAGELATCH_EX | 3874279 | 21546453 | 2617 | 1155926 |
| 2 | SOS_SCHEDULER_YIELD | 66572 | 8903310 | 6971 | 8903215 |
| 3 | BROKER_TASK_STOP | 306 | 1800370 | 11919 | 2497 |
| 4 | HADR_WORK_QUEUE | 162 | 1785334 | 15720 | 183694 |
| 5 | DIRTY_PAGE_POLL | 16184 | 1784054 | 352 | 41 |
| 6 | LOGMGR_QUEUE | 26332 | 1783863 | 398 | 346 |

3. A better way is to take multiple snapshots of the waitstats DMV in order to summarize the waits that occurred only during that period. Execute the following query (found in the WaitStats.sql script) in Management Studio in order to create a temporary table that contains two snapshots from sys.dm_os_wait_stats separated by a one minute delay:

```
-- Example of taking snapshots one minute apart.
select getdate() as Runtime, * into #temp from
sys.dm_os_wait_stats
go
waitfor delay '00:01:00'
go
insert into [#temp]
(Runtime,wait_type,waiting_tasks_count,wait_time_ms,max_wait_time
_ms,signal_wait_time_ms)
select getdate() as
RunTime,wait_type,waiting_tasks_count,wait_time_ms,max_wait_time_
ms,signal_wait_time_ms from sys.dm_os_wait_stats
go
```

With the 2 snapshots in question, we now can determine the waits that occurred during the period between the snapshots with a simple query. The query below coalesces some of the common wait types into groups, which can give you a gist of the primary bottleneck. All the lock wait types for example are coalesced into one LOCK group, and it also ignores some system wait types which in general we shouldn't worry about. You can get a more granular report by changing the CTE to remove the CASE statement that is grouping the wait types. Execute the following query (found in the WaitStats.sql script) in Management Studio to display the summarized wait types during the one minute period captured in the previous step. You will need to execute this query in the same window as the previous query in order to have access to the temporary table that was created:

```
--- This query will give you the difference in the Waitstats from
max snapshot tot he Min snapshot
SELECT MAX(runtime) as StarTime,MIN(runtime) as EndTime,
datediff(second,min(runtime),max(runtime)) as Diff_in_seconds
FROM #temp

Print '**** Server-level waitstats during the data capture
*******'
Print '';
```

```
WITH WaitCategoryStats (runtime, wait_category, wait_type,
wait_time_ms, waiting_tasks_count, max_wait_time_ms) AS
( SELECT runtime,
  CASE
      WHEN wait_type LIKE 'LCK%' THEN 'LOCKS'
        WHEN wait_type LIKE 'PAGEIO%' THEN 'PAGE I/O LATCH'
        WHEN wait_type LIKE 'PAGELATCH%' THEN 'PAGE LATCH (non-
I/O)'
        WHEN wait_type LIKE 'LATCH%' THEN 'LATCH (non-buffer)'
        WHEN wait_type LIKE 'LATCH%' THEN 'LATCH (non-buffer)'
        ELSE wait_type
  END AS wait_category, wait_type, wait_time_ms,
waiting_tasks_count, max_wait_time_ms
FROM #temp
)
SELECT TOP 15
  wait_category
    , MAX(wait_time_ms) - MIN(wait_time_ms) as wait_time_ms
    , (MAX(wait_time_ms) - MIN(wait_time_ms)) / (1 + datediff (s,
MIN(runtime), MAX(runtime))) as wait_time_ms_per_sec
      , MAX(waiting_tasks_count) max_waiting_tasks
      , (MAX(wait_time_ms) - MIN(wait_time_ms))/ Case
(MAX(waiting_tasks_count) - MIN(waiting_tasks_count))
           WHEN 0 THEN 1 ELSE ((MAX(waiting_tasks_count) -
MIN(waiting_tasks_count)))
         END AS average_wait_time_ms
      , MAX(max_wait_time_ms) AS max_Wait_time_ms
FROM WaitCategoryStats
WHERE runtime IN ( (SELECT MAX(runtime) from #temp),(SELECT
MIN(runtime) FROM #temp))
 AND wait_type NOT IN ('WAITFOR', 'LAZYWRITER_SLEEP',
'SQLTRACE_BUFFER_FLUSH', 'CXPACKET', 'EXCHANGE',
    'REQUEST_FOR_DEADLOCK_SEARCH', 'KSOURCE_WAKEUP',
'BROKER_TRANSMITTER', 'BROKER_EVENTHANDLER',
'ONDEMAND_TASK_QUEUE',
    'CHKPT', 'DBMIRROR_WORKER_QUEUE', 'DBMIRRORING_CMD',
'DBMIRROR_DBM_EVENT', 'XE_DISPATCHER_WAIT',
'FT_IFTS_SCHEDULER_IDLE_WAIT',
      'ASYNC_NETWORK_IO', 'PREEMPTIVE_OS_WAITFORSINGLEOBJECT',
'DIRTY_PAGE_POLL', 'LOGMGR_QUEUE',
'SQLTRACE_INCREMENTAL_FLUSH_SLEEP',
      'XE_TIMER_EVENT', 'CHECKPOINT_QUEUE',
'HADR_FILESTREAM_IOMGR_IOCOMPLETION', 'SLEEP_TASK',
'BROKER_TO_FLUSH', 'SOS_SCHEDULER_YIELD')
GROUP BY wait_category
ORDER BY wait_time_ms_per_sec DESC
```

Results | Messages

| | StarTime | EndTime | Diff_in_seconds |
|---|---|---|---|
| 1 | 2012-01-29 15:17:13.633 | 2012-01-29 15:16:13.587 | 60 |

| | wait_category | wait_time_ms | wait_time_ms_per_sec | max_waiting_tasks | average_wait_time_ms | max_Wait_time_ms |
|---|---|---|---|---|---|---|
| 1 | PAGE LATCH (non-I/O) | 29104107 | 477116 | 5153425 | 5 | 6546 |
| 2 | SOS_SCHEDULER_YIELD | 515730 | 8454 | 80979 | 100 | 6971 |
| 3 | CHECKPOINT_QUEUE | 92331 | 1513 | 996 | 46165 | 169961 |
| 4 | LATCH (non-buffer) | 69192 | 1134 | 277 | 249 | 2174 |
| 5 | BROKER_TASK_STOP | 64652 | 1059 | 381 | 5877 | 11919 |

4. We now know that PAGE_LATCH is our problem, but we need some further detail as to what this latch is on, and which queries are contributing to this bottleneck. There are a couple DMVs that can help us with that.

   First we can use sys.dm_os_waiting_tasks , which gives us every waiting task along with its wait type. This is a point in time query, so you will get waits ONLY if they are currently occurring. For some wait types such as PAGEIOLATCH and PAGELATCH the waits are rather short individually in spite of cumulatively being long, so you may or may not see individual sessions waiting.

```
select
session_id,wait_type,wait_duration_ms,resource_description,blocki
ng_session_id,*
from sys.dm_os_waiting_tasks
where wait_type like 'PAGELATCH%'
```

Results | Messages

| | session_id | wait_type | wait_duration_ms | resource_description | blocking_session_id |
|---|---|---|---|---|---|
| 1 | 53 | PAGELATCH_EX | 70 | 2:1:257 | NULL |
| 2 | 55 | PAGELATCH_EX | 76 | 2:1:257 | NULL |
| 3 | 54 | PAGELATCH_EX | 1 | 2:1:257 | NULL |
| 4 | 57 | PAGELATCH_EX | 72 | 2:1:257 | NULL |
| 5 | 58 | PAGELATCH_EX | 71 | 2:1:257 | NULL |

   Another way of looking at the queries that are waiting is to use sys.dm_exec_requests as was done in an earlier exercise.

```
select
a.session_id,
start_time,
b.program_name,
a.status,
blocking_session_id,
wait_type,
wait_time,
wait_resource,
 (SELECT TOP 1 SUBSTRING(s2.text,statement_start_offset / 2+1 ,
       ( (CASE WHEN statement_end_offset = -1
                  THEN (LEN(CONVERT(nvarchar(max),s2.text)) * 2)
                                    ELSE statement_end_offset END)
- statement_start_offset) / 2+1))  AS sql_statement

                                        , s2.text
```

```
,a.cpu_time,
a.Total_elapsed_time,
a.reads,
a.writes
from
sys.dm_exec_requests  a inner join
sys.dm_exec_sessions b on a.session_id = b.session_id
      CROSS APPLY sys.dm_exec_sql_text(a.sql_handle) AS s2
```

Examining the results reveals that there is definitely latch contention. We can identify the sessions in question, and in this case the contention is over the page 2:1:116 (you may have a different page in your results). Also we can see both sessions are allocating *temporary* tables.

Results | Messages

| | sessi... | start_time | program_n... | status | blocking_session_id | wait_type | wait_... | wait_res... | sql_statement | text |
|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 65 | 2012-01-29 15:36:06.223 | SQLCMD | suspended | 69 | PAGELATCH_EX | 94 | 2:1:120 | SELECT 1 AS c1 INTO #t1; | CREATE PROC usp_DoSomeWork AS |
| 15 | 66 | 2012-01-29 15:36:06.223 | SQLCMD | suspended | 69 | PAGELATCH_EX | 94 | 2:1:120 | SELECT 1 AS c1 INTO #t1; | CREATE PROC usp_DoSomeWork AS |
| 16 | 67 | 2012-01-29 15:36:06.223 | SQLCMD | suspended | 69 | PAGELATCH_EX | 92 | 2:1:120 | EXEC usp_DoSomeWork | DECLARE @x int SET @x = 1 WHIL |
| 17 | 68 | 2012-01-29 15:36:06.223 | SQLCMD | suspended | 69 | PAGELATCH_EX | 18 | 2:1:120 | SELECT 1 AS c1 INTO #t1; | CREATE PROC usp_DoSomeWork AS |
| 18 | 69 | 2012-01-29 15:36:06.227 | SQLCMD | runnable | 0 | NULL | 0 | 2:1:120 | EXEC usp_DoSomeWork | DECLARE @x int SET @x = 1 WHIL |
| 19 | 70 | 2012-01-29 15:36:06.223 | SQLCMD | suspended | 69 | PAGELATCH_EX | 94 | 2:1:120 | SELECT 1 AS c1 INTO #t1; | CREATE PROC usp_DoSomeWork AS |

5. Given the wait_resource for a latch is in this case a page, in order to figure out which object the contention is on, we either rely on the query or we can dump that page and see which object the page belongs to. You can take the wait_resource which is 2:1:116 and use that information to dump the page using the following query in Management Studio (it will be easier to read the output of this command if you switch the query results to Text via the menu option Query -> Results To -> Results to Text or by pressing Ctrl-T). If you got a different page number in your wait_resource column, be sure to replace "116" with the page number from your results:

```
dbcc traceon(3604,-1)
-- Displays the contents of the given page
-- Pages are addressed by database_id:file_id:page_id
-- i.e. 2:1:116 is the 116th page in the first file in tempdb
(which is always database_id 2)
dbcc page ( 2,1,116,3)
```

```
m_pageId = (1:120)                  m_headerVersion = 1                 m_type = 2
m_typeFlagBits = 0x4                m_level = 0                         m_flagBits = 0x0
m_objId (AllocUnitId.idObj) = 34    m_indexId (AllocUnitId.idInd) = 3   Metadata: AllocUnitId = 844424932360192
Metadata: PartitionId = 844424932360192                                  Metadata: IndexId = 3
Metadata: ObjectId = 34             m_prevPage = (0:0)                  m_nextPage = (1:257)
pminlen = 10                        m_slotCnt = 31                      m_freeCnt = 4900
```

**Note:** DBCC PAGE is not an officially documented command. It is "lightly" documented in that it does appear in some official Microsoft Knowledgebase articles, but you do not want to rely on this command or its output in any applications. It is simply used for illustration purposes here.

From that output you get the object ID and you can then find the object which turns out to be a system table sysschobjs.

```
USE tempdb
GO
select object name(34)
GO
USE AdventureWorksPTO
GO
```

The sysschobjs table contains a row for each object. At this point, given the fact that you are not directly using the system table, all you can do is from the queries figure out why you are creating temp tables at such a rapid rate which in turn is causing contention on system tables. That is more of a logical application design question and a review of the stored procedure will be warranted to see if there is a single code path where we can reduce temp table creation and the associated contention will then disappear.

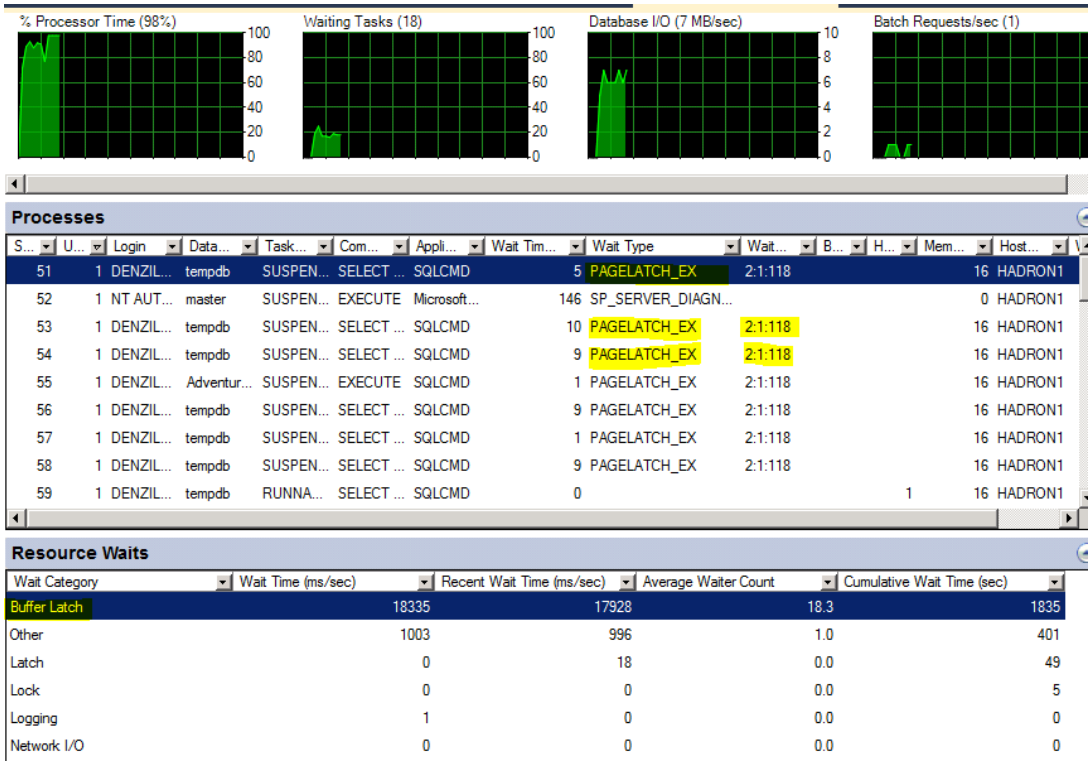
6. This does not have to be a system table as in this example, it can also be a user table. Another DMV is very helpful in the latch cases, in particular if the latch is not a buffer latch, but rather a LATCH_XX. Once again when looking at this DMV you should take snapshots otherwise the waits are cumulative since the last server restart.

```
select * from sys.dm_os_latch_stats
order by wait_time_ms desc
```

Results | Messages

| | latch_class | waiting_requests_count | wait_time_ms | max_wait_time_ms |
|---|---|---|---|---|
| 1 | BUFFER | 3345203 | 18621885 | 2618 |
| 2 | LOG_MANAGER | 252 | 68586 | 2175 |
| 3 | FGCB_ADD_REMOVE | 51 | 1967 | 99 |
| 4 | ACCESS_METHODS_HOBT_VIRTUAL_ROOT | 70 | 1601 | 77 |
| 5 | ACCESS_METHODS_HOBT_COUNT | 4 | 102 | 51 |

7. You can do the same analysis via the Activity Monitor. If you look at the highlighted section under processes you can see the wait types. The cumulative wait_type can also be seen from the Resource Waits section of Activity Monitor. It holds a weighted average calculated over recent history, but doesn't store any long term history. You can get details of the session in question and the statement that is running as well.




| S... | U... | Login | Data... | Task... | Com... | Appli... | Wait Tim... | Wait Type | Wait... | B... | H... | Mem... | Host... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 51 | 1 | DENZIL... | tempdb | SUSPEN... | SELECT ... | SQLCMD | 5 | PAGELATCH_EX | 2:1:118 | | | 16 | HADRON1 |
| 52 | 1 | NT AUT... | master | SUSPEN... | EXECUTE | Microsoft... | 146 | SP_SERVER_DIAGN... | | | | 0 | HADRON1 |
| 53 | 1 | DENZIL... | tempdb | SUSPEN... | SELECT ... | SQLCMD | 10 | PAGELATCH_EX | 2:1:118 | | | 16 | HADRON1 |
| 54 | 1 | DENZIL... | tempdb | SUSPEN... | SELECT ... | SQLCMD | 9 | PAGELATCH_EX | 2:1:118 | | | 16 | HADRON1 |
| 55 | 1 | DENZIL... | Adventur... | SUSPEN... | EXECUTE | SQLCMD | 1 | PAGELATCH_EX | 2:1:118 | | | 16 | HADRON1 |
| 56 | 1 | DENZIL... | tempdb | SUSPEN... | SELECT ... | SQLCMD | 9 | PAGELATCH_EX | 2:1:118 | | | 16 | HADRON1 |
| 57 | 1 | DENZIL... | tempdb | SUSPEN... | SELECT ... | SQLCMD | 1 | PAGELATCH_EX | 2:1:118 | | | 16 | HADRON1 |
| 58 | 1 | DENZIL... | tempdb | SUSPEN... | SELECT ... | SQLCMD | 9 | PAGELATCH_EX | 2:1:118 | | | 16 | HADRON1 |
| 59 | 1 | DENZIL... | tempdb | RUNNA... | SELECT ... | SQLCMD | 0 | | | | 1 | 16 | HADRON1 |

| Wait Category | Wait Time (ms/sec) | Recent Wait Time (ms/sec) | Average Waiter Count | Cumulative Wait Time (sec) |
|---|---|---|---|---|
| Buffer Latch | 18335 | 17928 | 18.3 | 1835 |
| Other | 1003 | 996 | 1.0 | 401 |
| Latch | 0 | 18 | 0.0 | 49 |
| Lock | 0 | 0 | 0.0 | 5 |
| Logging | 1 | 0 | 0.0 | 0 |
| Network I/O | 0 | 0 | 0.0 | 0 |

8. Once you have finished reviewing the results of the queries and Activity Monitor, return to the Command Prompt and press Enter to end the simulation and clean up. Close Activity Monitor and any queries you have open in Management Studio.

# Exercise 3: Investigating Memory clerks and memory usage

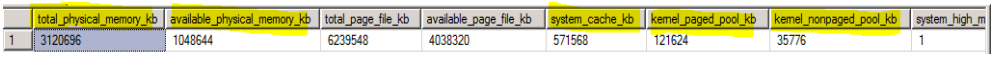
## Objectives

- Learn to understand how to account for SQL Server memory
- Use DMV's in order to help identify a caches that consume memory
- Use XEvents in order to aid in identifying memory allocations

## Task: Explore memory related DMVs and account for SQL Server memory allocations.

1. How much memory is available on the system? You can address this from Performance Monitor, but there is a new DMV available that gives you this information directly from SQL Server. You can find the following queries in the MemoryDMVs.sql script file found in C:\Labs\Module1\LabFiles\Exercise3:

```
select * from sys.dm_os_sys_memory
```



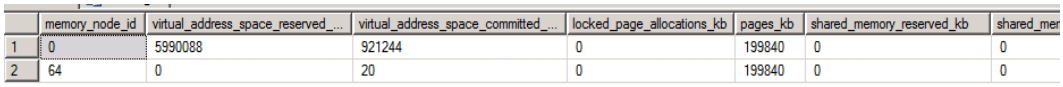
| | total_physical_memory_kb | available_physical_memory_kb | total_page_file_kb | available_page_file_kb | system_cache_kb | kernel_paged_pool_kb | kernel_nonpaged_pool_kb | system_high_m |
|---|---|---|---|---|---|---|---|---|
| 1 | 3120696 | 1048644 | 6239548 | 4038320 | 571568 | 121624 | 35776 | 1 |

Also at the process level:

```
select * from sys.dm_os_process_memory
```

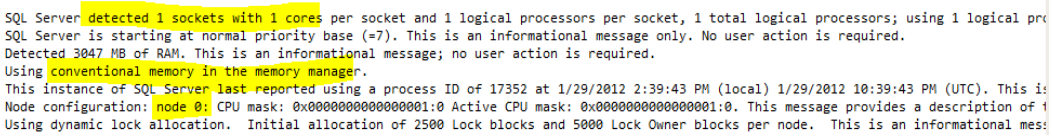
2. Is this a NUMA Machine? You will see multiple memory nodes starting at node 0 along with the memory allocated to each node. The foreign_commited_kb also indicates how much of this memory is committed from remote nodes. Accessing remote memory is far slower than local memory on a NUMA box.
Note: Node 64 is the DAC (dedicated admin) node.

```
select * from sys.dm_os_memory_nodes
```



| | memory_node_id | virtual_address_space_reserved_... | virtual_address_space_committed_... | locked_page_allocations_kb | pages_kb | shared_memory_reserved_kb | shared_mer |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 5990088 | 921244 | 0 | 199840 | 0 | 0 |
| 2 | 64 | 0 | 20 | 0 | 199840 | 0 | 0 |

Another way to see NUMA is to look at the errorlog. If you see more than 1 node, it is a NUMA machine. You will also see the memory mode, number of sockets and cores etc.



```
SQL Server detected 1 sockets with 1 cores per socket and 1 logical processors per socket, 1 total logical processors; using 1 logical pr
SQL Server is starting at normal priority base (=7). This is an informational message only. No user action is required.
Detected 3047 MB of RAM. This is an informational message; no user action is required.
Using conventional memory in the memory manager.
This instance of SQL Server last reported using a process ID of 17352 at 1/29/2012 2:39:43 PM (local) 1/29/2012 10:39:43 PM (UTC). This i
Node configuration: node 0: CPU mask: 0x0000000000000001:0 Active CPU mask: 0x0000000000000001:0. This message provides a description of
Using dynamic lock allocation.  Initial allocation of 2500 Lock blocks and 5000 Lock Owner blocks per node.  This is an informational mes
```

3. Look at the DMV sys.dm_os_memory_brokers, and you see 3 brokers below as well as if their trend is currently GROW or SHRINK, and you see if overall Caches are consuming a lot OR Stolen memory is high.

```
select * from sys.dm_os_memory_brokers
```

| Value | Description |
|---|---|
| MEMORYBROKER_FOR_CACHE | Memory allocated for use by cached objects. |
| MEMORYBROKER_FOR_STEAL | Memory stolen from buffer pool. This memory is not available for re-use by other components until freed by the current owner. |
| MEMORYBROKER_FOR_RESERVE | This is memory reserved for future use by currently executing requests. |

4. Open a Command Prompt and run the following script which will bloat the procedure cache. We will use DMVs in order to help figure out where the bloat is coming from.

\Labs\Module1\LabFiles\Exercise3\CacheBloat\Scenario.cmd

5. Give this repro about 60-120 seconds of running time before we actually start investigating the problem. First we will start with the Waits and Queues methodology. Ideally when looking at wait stats, we ought to take multiple snapshots and calculate the difference as in Exercise1, however for convenience wait stats were cleared when we started the report so the DMV now contains only the current wait stats.

```
select * from sys.dm_os_wait_stats
order by wait_time_ms desc
```

Results | Messages

| | wait_type | waiting_tasks_count | wait_time_ms | max_wait_time_ms | signal_wait_time_ms |
|---|---|---|---|---|---|
| 1 | EE_PMOLOCK | 8526 | 219779 | 468 | 23321 |
| 2 | CLR_AUTO_EVENT | 132 | 154664 | 76869 | 80 |
| 3 | HADR_WORK_QUEUE | 53 | 38962 | 1153 | 23 |
| 4 | BROKER_TASK_STOP | 41 | 27727 | 5061 | 107 |

Note that you may see the CLR_AUTO_EVENT wait type at the top of the list. This is a system wait type related to CLR code execution and can typically be safely ignored. There are several wait types to do with memory, some of the more common ones are found below.

(The full list can be found at: http://msdn.microsoft.com/en-us/library/ms179984.aspx)

| | |
|---|---|
| EE_PMOLOCK | Occurs during synchronization of certain types of memory allocations during statement execution. |
| RESOURCE_SEMAPHORE | Occurs when a query memory request cannot be granted immediately due to other concurrent queries. High waits and wait times may indicate excessive number of concurrent queries, or excessive memory request amounts. |
| RESOURCE_SEMAPHORE_SMALL_QUERY | Occurs when memory request by a small query cannot be granted immediately due to other concurrent queries. Wait time should not exceed more than a few seconds, because the server transfers the request to the main query memory pool if it fails to grant the requested memory within a few seconds. High waits may indicate an excessive number of concurrent small queries while the main memory pool is blocked by waiting queries. |
| SOS_VIRTUALMEMORY_LOW | Occurs when a memory allocation waits for a resource manager to free up virtual memory. |
| CMEMTHREAD | Occurs when a task is waiting on a thread-safe memory object. The wait time might increase when there is contention caused by multiple tasks trying to allocate memory from the same memory object. |

6. We now know that the primary wait type is related to some sort of memory pressure. Often in the case of memory issues, it may be a memory allocation error that leads us to investigate the memory health on the box rather than a wait type.

```
select * from sys.dm_os_memory_clerks
order by pages_kb desc
```

Results | Messages

| | type | name | memory_node_id | pages_kb | virtual_memory_reserved_kb | virtual_memory_committed_kb | awe_allocated_kb |
|---|---|---|---|---|---|---|---|
| 1 | CACHESTORE_SQLCP | SQL Plans | 0 | 566232 | 0 | 0 | 0 |
| 2 | MEMORYCLERK_SQLBUFFERPOOL | Default | 0 | 97408 | 1207628 | 6876 | 0 |
| 3 | MEMORYCLERK_SOSNODE | SOS_Node | 0 | 32072 | 0 | 0 | 0 |
| 4 | MEMORYCLERK_SQLSTORENG | Default | 0 | 6584 | 4800 | 4800 | 0 |
| 5 | MEMORYCLERK_SQLGENERAL | Default | 0 | 6216 | 0 | 0 | 0 |
| 6 | OBJECTSTORE_XACT_CACHE | Transactions | 0 | 5600 | 0 | 0 | 0 |
| 7 | MEMORYCLERK_SQLCLR | Default | 0 | 5320 | 1604608 | 34032 | 0 |

**Note**: In previous versions of the product, each of the clerks had separate entries for single_pages_kb and multi_pages_kb. With the memory manager redesign in SQL 2012 they are now consolidated into pages_kb.

7. To become familiar with what the various memory caches are, you can query the DMV below:

```
SELECT  TOP 10
  LEFT([name], 20) as [name],
  LEFT([type], 20) as [type],
  pages_kb AS cache_kb,
  [entries_count]
FROM sys.dm_os_memory_cache_counters
order by pages_kb DESC
```

| | name | type | cache_kb | entries_count |
|---|---|---|---|---|
| 1 | SQL Plans | CACHESTORE_SQLCP | 584688 | 1333 |
| 2 | Bound Trees | CACHESTORE_PHDR | 7776 | 99 |
| 3 | mssqlsystemresource | USERSTORE_DBMETADATA | 4792 | 2142 |
| 4 | SchemaMgr Store | USERSTORE_SCHEMAMGR | 2680 | 0 |
| 5 | msdb | USERSTORE_DBMETADATA | 2608 | 1227 |
| 6 | Object Plans | CACHESTORE_OBJCP | 1336 | 3 |

8. Given we now know for a fact that the cache that is bloated is the procedure cache (i.e. one that holds the SQL Query plans) we can use the plan cache DMVs in order to figure out what is polluting the cache.

   We use a concept called a query fingerprint (available in the query_hash column in sys.dm_exec_requests and sys.dm_exec_query_stats) in order to identify unique queries in the cache. A query hash basically refers to the "normalized" form of the

query. The hash for the 2 queries below is the same even though they have different literal values:

```
SELECT * FROM Person.BusinessEntity c
WHERE c.BusinessEntityID IN (1900, 1500)
GO
SELECT * FROM Person.BusinessEntity c
WHERE c.BusinessEntityID IN (1901, 1501)
GO
```

Given that, using that query_hash column we can figure out if the cache pollution is due to ad-hoc statements that need to be parameterized.

```
select query_hash,count(*) as DistinctQueriesOfSameType,
sum(CAST(size_in_bytes AS BIGINT)) as TotalBytes,
 sum(usecounts) as SumUseCounts
 from sys.dm_exec_cached_plans a
inner join sys.dm_exec_query_stats b on a.plan_handle =
b.plan_handle
group by query_hash
order by TotalBytes desc
```

Results | Messages

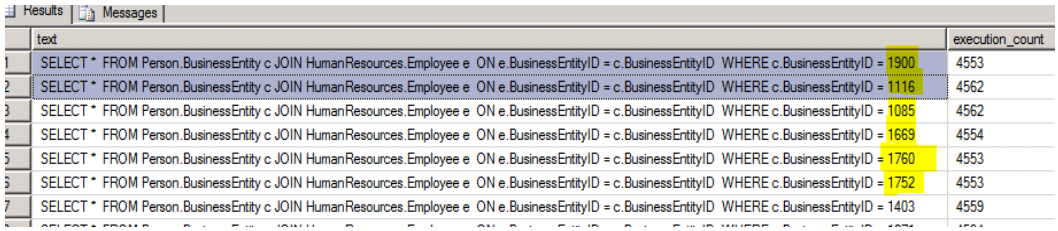
| | query_hash | DistinctQueriesOfSameType | TotalBytes | SumUseCounts |
|---|---|---|---|---|
| 1 | 0xAAB6B7E2C4E0844D | 1154 | 569827328 | 3397710 |
| 2 | 0x5FDBC10750B792B3 | 22 | 8552448 | 3396 |
| 3 | 0x3B69C5192A854844 | 70 | 3440640 | 150 |
| 4 | 0xEEF6EE7EB07915B8 | 6 | 1892352 | 4635 |
| 5 | 0x39EB32B302DB7CBA | 1 | 1359872 | 18 |
| 6 | 0xD3BB2A144F128171 | 10 | 1064960 | 12 |
| 7 | 0xFDA84BB0554C256B | 1 | 811008 | 114 |
| 8 | 0x994F7A04F53BC2AC | 1 | 794624 | 1 |

Now let us see what query results from that query_hash

```
-- Figure out what query needs to be parameterized
-- This is an adhoc query that needs to be parameterized
select S2.text,a.execution_count from sys.dm_exec_query_stats a
cross apply sys.dm_exec_sql_text(sql_handle) as S2
where a.query_hash = 0xAAB6B7E2C4E0844D
```

This definitely looks like a case of Adhoc SQL that is polluting the cache. The only thing different between these queries is the literal value at the end of the statement. If we were

to parameterize this, we would improve performance, reduce the Procedure Cache bloat and benefit overall.



Results | Messages

| | text | execution_count |
|---|---|---|
| 1 | SELECT * FROM Person.BusinessEntity c JOIN HumanResources.Employee e ON e.BusinessEntityID = c.BusinessEntityID WHERE c.BusinessEntityID = 1900 | 4553 |
| 2 | SELECT * FROM Person.BusinessEntity c JOIN HumanResources.Employee e ON e.BusinessEntityID = c.BusinessEntityID WHERE c.BusinessEntityID = 1116 | 4562 |
| 3 | SELECT * FROM Person.BusinessEntity c JOIN HumanResources.Employee e ON e.BusinessEntityID = c.BusinessEntityID WHERE c.BusinessEntityID = 1085 | 4562 |
| 4 | SELECT * FROM Person.BusinessEntity c JOIN HumanResources.Employee e ON e.BusinessEntityID = c.BusinessEntityID WHERE c.BusinessEntityID = 1669 | 4554 |
| 5 | SELECT * FROM Person.BusinessEntity c JOIN HumanResources.Employee e ON e.BusinessEntityID = c.BusinessEntityID WHERE c.BusinessEntityID = 1760 | 4553 |
| 6 | SELECT * FROM Person.BusinessEntity c JOIN HumanResources.Employee e ON e.BusinessEntityID = c.BusinessEntityID WHERE c.BusinessEntityID = 1752 | 4553 |
| 7 | SELECT * FROM Person.BusinessEntity c JOIN HumanResources.Employee e ON e.BusinessEntityID = c.BusinessEntityID WHERE c.BusinessEntityID = 1403 | 4559 |

## Task: Explore the memory Allocations using Extended Events

1. Ensure that same Procedure Cache Bloat scenario is running. If you stopped it, start it up again

   \Labs\Module1\LabFiles\Exercise3\CacheBloat\Scenario.cmd

2. We are not going to "cover" Extended Events here, we are only going to expose the fact that there exist XEvents that will help in troubleshooting memory allocation type issues.

   Run the following script to create an Extended Event session. The code below can be found in the CreateExtendedEventSession.sql script in C:\Labs\Module1\LabFiles\Exercise3. Specifically we use the "SQLOS.Page_allocated" and "SQLOS.Page_freed" events, and are filtering ONLY for the CACHESTORE_SQLCP which we know is the root of our problem. Run the script below to create the extended event

```
CREATE EVENT SESSION [MemoryXE] ON SERVER
ADD EVENT sqlos.allocation_failure(

ACTION(package0.callstack,sqlos.worker_address,sqlserver.clien
t_app_name,sqlserver.query_hash,sqlserver.session_id,sqlserver
.sql_text,sqlserver.tsql_stack)),
ADD EVENT sqlos.page_allocated(

ACTION(package0.callstack,sqlserver.query_hash,sqlserver.query
_plan_hash,sqlserver.session_id,sqlserver.sql_text,sqlserver.t
sql_frame,sqlserver.tsql_stack)
    WHERE ([memory_clerk_name]=N'CACHESTORE_SQLCP')),
ADD EVENT sqlos.page_freed(

ACTION(package0.callstack,sqlserver.client_app_name,sqlserver.
query_hash,sqlserver.query_plan_hash,sqlserver.session_id,sqls
erver.sql_text)
```

```
    WHERE ([memory_clerk_name]=N'CACHESTORE_SQLCP'))
ADD TARGET package0.event_file(SET filename=N'C:\Program
Files\Microsoft SQL
Server\MSSQL11.MSSQLSERVER\MSSQL\Log\MemoryXE.xel',max_rollove
r_files=(0))
WITH (MAX_MEMORY=4096
KB,EVENT_RETENTION_MODE=ALLOW_SINGLE_EVENT_LOSS,MAX_DISPATCH_L
ATENCY=30 SECONDS,MAX_EVENT_SIZE=0
KB,MEMORY_PARTITION_MODE=NONE,TRACK_CAUSALITY=OFF,STARTUP_STAT
E=OFF)
GO
```

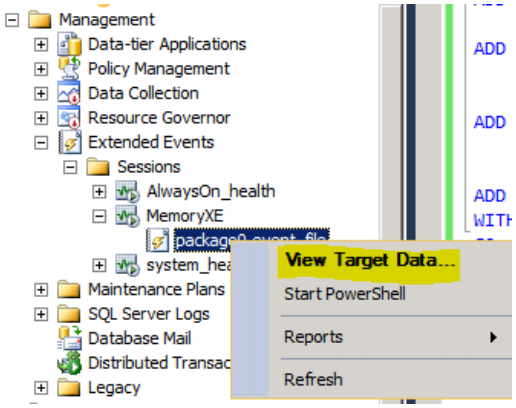
3. Once you have the Extended Event session created, you will be able to view it under Management -> Extended Events -> Sessions in Management Studio. You can start it here using the GUI, or with the following command:

```
Alter event session [MemoryXe] ON Server  State =START
```

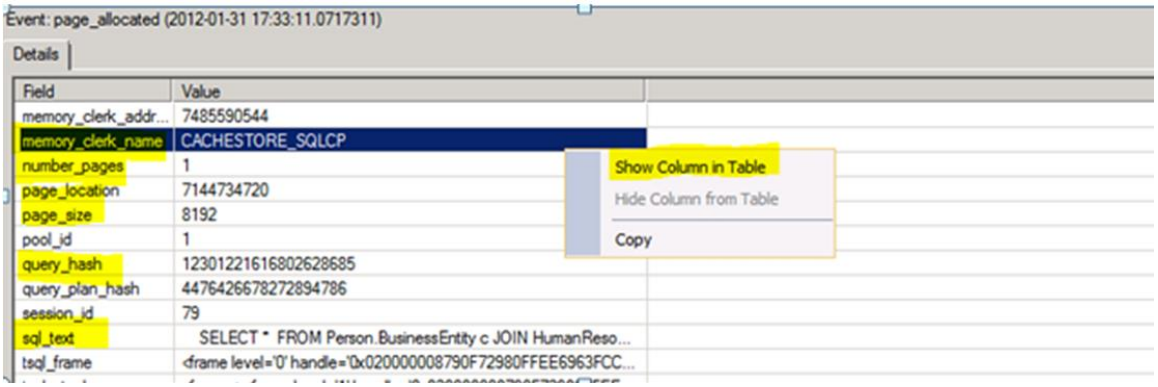
4. The memory allocation Extended Event can be fairly chatty, so give it about 60 seconds or so and then "stop" the collection of the event either with Management Studio or the following command:

```
Alter event session [MemoryXe] ON Server  State =STOP
```

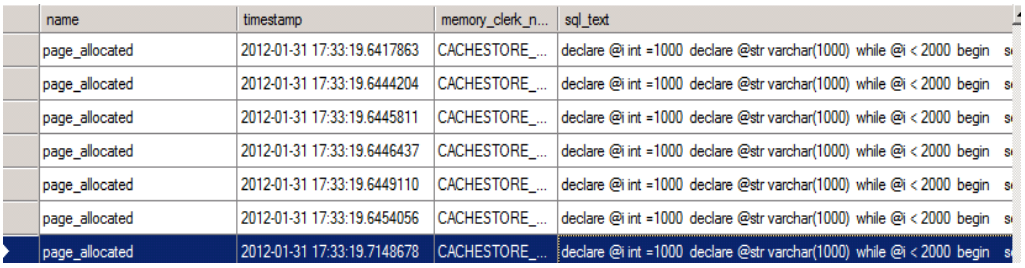
5. View the collected data by clicking on View Target Data as below:




6. Format the columns as below so that it is more readable. You can add the columns to the grid that is displayed at the top by right clicking on each of the columns highlighted below and clicking Show column in table.

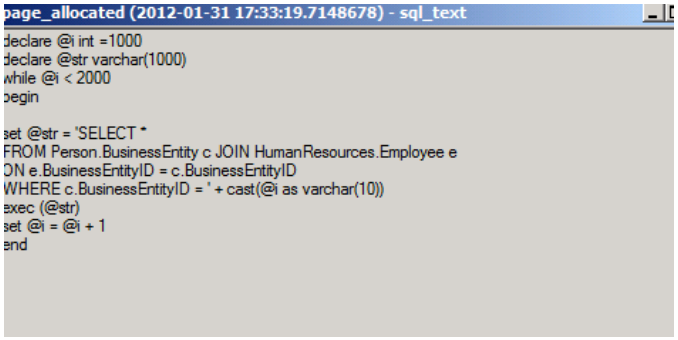



You will end up with a view such as this

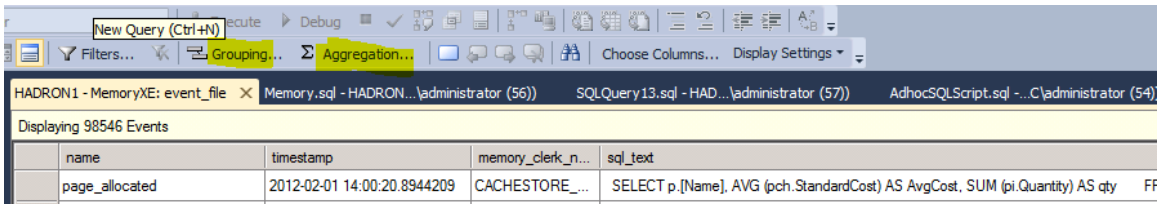


| name | timestamp | memory_clerk_n... | sql_text |
|---|---|---|---|
| page_allocated | 2012-01-31 17:33:19.6417863 | CACHESTORE_... | declare @i int =1000 declare @str varchar(1000) while @i < 2000 begin |
| page_allocated | 2012-01-31 17:33:19.6444204 | CACHESTORE_... | declare @i int =1000 declare @str varchar(1000) while @i < 2000 begin |
| page_allocated | 2012-01-31 17:33:19.6445811 | CACHESTORE_... | declare @i int =1000 declare @str varchar(1000) while @i < 2000 begin |
| page_allocated | 2012-01-31 17:33:19.6446437 | CACHESTORE_... | declare @i int =1000 declare @str varchar(1000) while @i < 2000 begin |
| page_allocated | 2012-01-31 17:33:19.6449110 | CACHESTORE_... | declare @i int =1000 declare @str varchar(1000) while @i < 2000 begin |
| page_allocated | 2012-01-31 17:33:19.6454056 | CACHESTORE_... | declare @i int =1000 declare @str varchar(1000) while @i < 2000 begin |
| page_allocated | 2012-01-31 17:33:19.7148678 | CACHESTORE_... | declare @i int =1000 declare @str varchar(1000) while @i < 2000 begin |

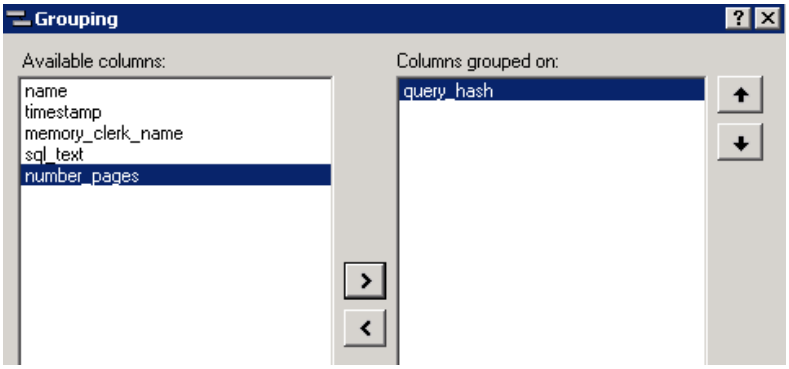
Double Clicking on the sql_text in the bottom pane will give you the whole batch which is a problem



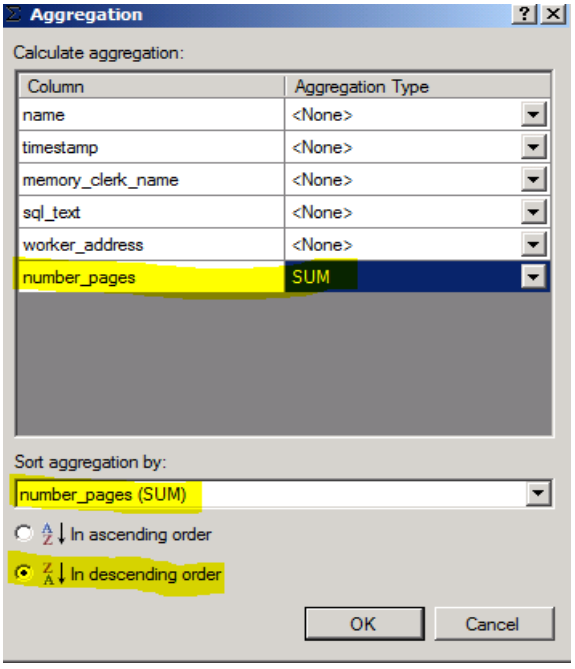

7. To get an aggregated view you can use the Grouping and Aggregation buttons in the Extended events toolbar



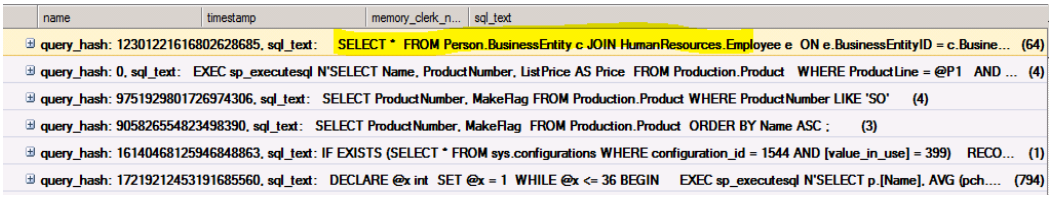
Click on the "Grouping" button in the toolbar (only available once you are viewing the Extended event data) and add the query_hash column as below:




Then Click on the "Aggregations" toolbar button and configure the options below




Now you get an aggregate view of who is doing the allocations, the top consumer being the same query we identified in the DMV section



This XEvent ( Pages_allocated and pages_freed) can be used to track memory leaks as well (ie: if you have pages that are allocated and not freed it is a leak) where you can get the callstack that is doing the allocation, the TSQL Statement and a bunch of other information.

8. Once you have finished reviewing the results of the queries, return to the Command Prompt and press Enter to end the simulation and clean up. Close any queries you have open in Management Studio.

**Note:** It is important to hit enter on the scenarios once they are done to run the cleanup batch file that cleans things up and resets them to the default values.