High_CPU_utilization_due_to_indexes_rebuild_on_Global_TempDB

Last updated by | Vitor Tomaz | Feb 24, 2023 at 3:27 AM PST

Contents

Issue

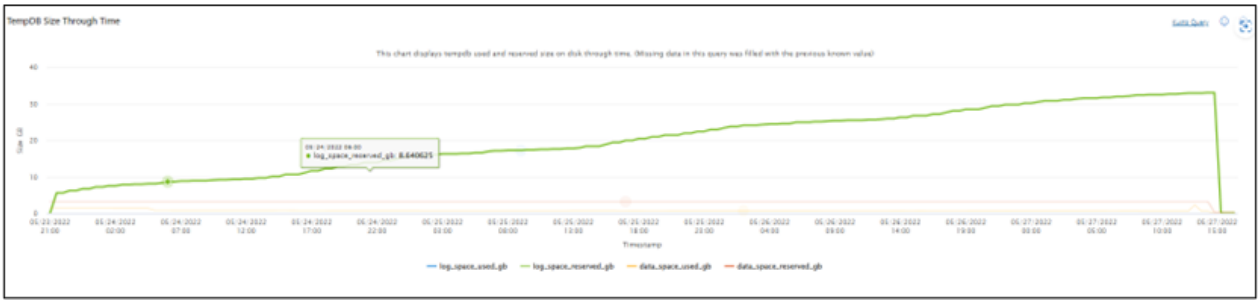
Customer has reported high CPU utilization which we can notice by the performance metrics.

Looking at the database performance, we can see corresponding to the high CPU, the TempDB log size was increasing. The issue appeared suddenly and there were no high consuming queries that were identified as the cause of high CPU, also the PVS was increasing.

You are advised to follow this TGS in case you have noticed open long transaction with the names **"sort_fake_worktable"** and **"sort_init"** while checking it on our telemetries or on the customer side using this DMV (*tempdb.sys.dm_tran_active_transactions*).

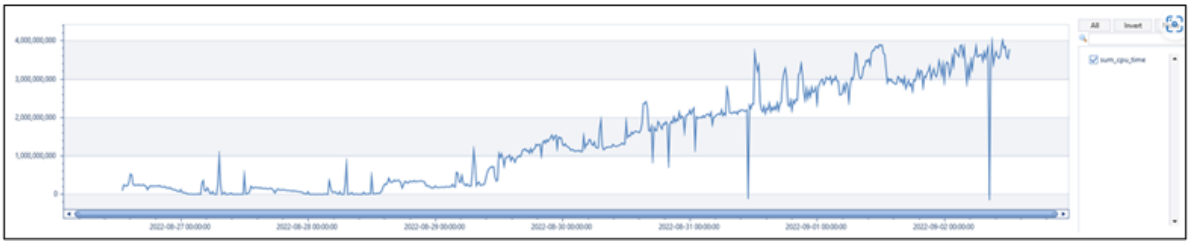
Investigation/Analysis

Checking the TempDB size on ASC (SQL troubleshooter -> Performance ->TempDB -> TempDB Size Through Time). We can notice the increase as below:



Checking CPU increase comparing it with the execution count MonWiQdsExecStats (QDS Execution Stats) - Overview (visualstudio.com)

```
MonWiQdsExecStats
| where TIMESTAMP > ago(7d)
| where LogicalServerName == "******" and database_name == "*****"
| summarize sum(cpu_time) by bin(TIMESTAMP, 15m)
```



We noticed that the CPU has increased while the execution count didn't increase.

To check open transactions on customer side, you can use the below T-SQL:

```
select * from tempdb.sys.dm_tran_active_transactions
```

For more information: sys.dm_tran_active_transactions (Transact-SQL) - SQL Server | Microsoft Learn

Within the opened transactions, we notice the names (**"sort_fake_worktable"** and **"sort_init"**) and session ID was -1.

Example

```
03795677    SELECT    8/28/2022 7:53:42 AM
503796849    sort_fake_worktable    8/28/2022 7:53:42 AM
503796850    sort_init    8/28/2022 7:53:42 AM
```

sort_fake_worktable and sort_init are internal transactions used by QP.

To check active sessions from CSS side: Active Transactions - Overview (visualstudio.com)

```
MonDmTranActiveTransactions
| where AppName == "*******"
| where TIMESTAMP > ago(2h)
```

transaction_id	database_id	user_db_name	transaction_begin_time	transaction_ty
503796850	-1		2022-08-28 09:53:42.7370000	
503797338	-1		2022-08-28 09:53:42.7530000	
503797914	-1		2022-08-28 09:53:42.7800000	
503798464	-1		2022-08-28 09:53:42.8100000	
503799010	-1		2022-08-28 09:53:42.8300000	
503799636	-1		2022-08-28 09:53:42.8500000	
503796850	-1		2022-08-28 09:53:42.7370000	

Note: You can also check the long running transaction in ASC (Performance > Blocking & Deadlocking).

To verify the type of transactions that were executing during the reported time, you can use the below Kusto query:

```
MonQueryProcessing
| where AppName !startswith 'b-' and AppName !startswith 'v-'
| where event == 'metadata_change'
| extend database_name=logical_database_name
| where LogicalServerName =~ '****' and database_name =~ '*****'
| project originalEventTimestamp, database_name, object_id, operation_type, operation_code
| order by originalEventTimestamp desc
```

Below you can see in the **operatiuon_type** column the transaction type as below:

object_id	operation_type	operation_code
41	CreateStats	0
41	CreateStats	0
203147769	CreateStats	0
41	CreateStats	0
41	CreateStats	0
5	AltDB	5
181575685	CreateStats	0
41	CreateStats	0
41	CreateStats	0
41	CreateStats	0
41	CreateStats	0

In this part, noticed that there were no rebuild indexes operation running but, by checking **MonIndexBuild** table we can notice there were index rebuild for the global tempDB, you can verify it using the below script:

```
MonIndexBuild
| where LogicalServerName =~ "******"
| where AppName =~ "******"
| where NodeName =~ "******"
| project originalEventTimestamp, LogicalServerName, event, logical_database_name, database_id, object_id
```

event	logical_database_name	database_id	object_id
create_index_event	pharmacydbv2	2	1644857968
create_index_event	pharmacydbv2	2	1676858082
create_index_event	pharmacydbv2	2	1708858196
create_index_event	pharmacydbv2	2	1740858310
create_index_event	pharmacydbv2	2	1772858424
create_index_event	pharmacydbv2	2	1804858538
create_index_event	pharmacydbv2	2	1836858652

The indexes rebuild time is near the same time of the long running transactions with sort_init transactions.

```
MonDmTranActiveTransactions
| where AppName == "*******"
| where TIMESTAMP between (datetime(yyyy-mm-dd 00:00:00)..datetime(yyyy-mm-dd 00:00:00))
| distinct transaction_begin_time, transaction_id, transaction_type, transaction_state, accessed_tempdb
```

transaction_begin_time	transaction_id	transaction_type	transaction_state	accessed_tempdb
2022-08-28 09:53:42.7370000	503796850	1	2	1
2022-08-28 09:53:42.7530000	503797338	1	2	1
2022-08-28 09:53:42.7800000	503797914	1	2	1
2022-08-28 09:53:42.8100000	503798464	1	2	1
2022-08-28 09:53:42.8300000	503799010	1	2	1
2022-08-28 09:53:42.8500000	503799636	1	2	1
2022-08-28 10:33:11.9900000	506401674	1	2	1
2022-08-28 10:33:11.9900000	506401678	1	2	1
2022-08-28 10:33:11.9900000	506401683	1	2	1
2022-08-28 10:33:11.9900000	506401686	1	2	1
2022-08-28 10:33:11.9970000	506401703	1	2	1
2022-08-28 10:33:12.0000000	506401715	1	2	1

sort_init is an internal transaction that is started as part of a rowset sort, most commonly as part of an index build.

It turns out that indexes were being created continuously on global temp tables. Also, checking the object ID captured you can notice that these were from GlobalTempDB and LocalTempDB.

```
1  create table ##globalTempTable (
2  id              int identity(1,1)
3  );
4
5  create table #localTempTable (
6  id              int identity(1,1)
7  );
8
9  select
10     *
11 from
12     [tempdb].sys.objects
13 order by
14     [name]
```

Results | Messages

	name	object_id
1	##globalTempTable	1061578820
2	#A6AD50DB	-1498591013
3	#localTempTable	-1164327821

Mitigation

From customer side we can use the below T-SQL script to check what transactions and sessions are creating log records in TempDB

```
select dst.session_id, dbt.transaction_id, dat.transaction_begin_time, dbt.database_transaction_begin_time, dbt.database_transaction_log_bytes_use
FROM tempdb.sys.dm_tran_database_transactions dbt, tempdb.sys.dm_tran_active_transactions dat, tempdb.sys.dm_tran_session_transactions dst
WHERE dst.transaction_id = dbt.transaction_id and dbt.transaction_id = dat.transaction_id and dbt.database_id = 2 and dbt.database_transaction_log
```

And you can use the below T-SQL as well to check the long running transactions:

```
DECLARE @request_duration_sec int
SET @request_duration_sec = 30
SELECT session_id, request_id, start_time, database_id FROM sys.dm_exec_requests
WHERE (open_transaction_count > 0 or open_resultset_count > 0) and datediff(second, start_time, getutcdate()) > @request_duration_sec
```

TempDB log gets full due to long active stats transactions. In summary, the root cause is that long-running index builds were being performed on hundreds of thousands of global temp, which gradually consumed more tempdb log space, PVS space, and CPU.

The customer needs to investigate from their side what process(es) were creating global temp tables and building indexes on them from the reported time frame.

Internal Reference

- Incident-332209397 Details - IcM (microsofticm.com)
- Incident-224946515 Details - IcM (microsofticm.com)

Root Cause Classification

Azure/SQL Database/Performance and Query Execution/Managing space for databases and pools

How good have you found this content?