

Plan_regression

Last updated by | Peter Hewitt | Sep 28, 2022 at 12:58 AM PDT

Contents

- [Issue](#)
- [Investigation/Analysis](#)
 - [Using Query store](#)
 - [Sample Troubleshooting](#)
 - [The example output shows that:](#)
- [Mitigation](#)
- [Public Doc Reference](#)
- [Root Cause Classification](#)

Issue

If a customer reports High CPU utilisation one of the potential causes can be SQL Execution Plan regression.

Background: To execute queries, the SQL Server Database Engine must analyse the statement to determine the most efficient way to access the required data. This analysis is handled by the Query Optimiser. The input to the Query Optimiser consists of the query, the database schema (table and index definitions), and the database statistics. The output of the Query Optimiser is a query execution plan. Plan regression happens when the SQL Server Query Optimiser compiles a new execution plan for a query which is sub-optimal/worse performing than the previous plan, in turn this can increase CPU time and query duration. Plan regression can occur for a number of reasons, including: the volume of rows inside a table has changed, change in table schema, removal/addition of indexes, out of date statistics.

This TSG will help you identify if plan regression is involved in contributing to the high CPU utilisation and the techniques to help mitigate the issue.

Investigation/Analysis

Identify which queries are the top consumers of CPU. This can be achieved by using the Performance > Queries reports in ASC or alternatively, with the below Kusto query:

```
MonWiQdsExecStats
| where LogicalServerName == "<>"
| where database_name == "<>"
| where TIMESTAMP > ago(1d)
| summarize sum(cpu_time) by bin(TIMESTAMP, 15m), query_hash
| order by sum_cpu_time desc
```

To understand the wait related information for a particular Query Hash, use the below Kusto query:

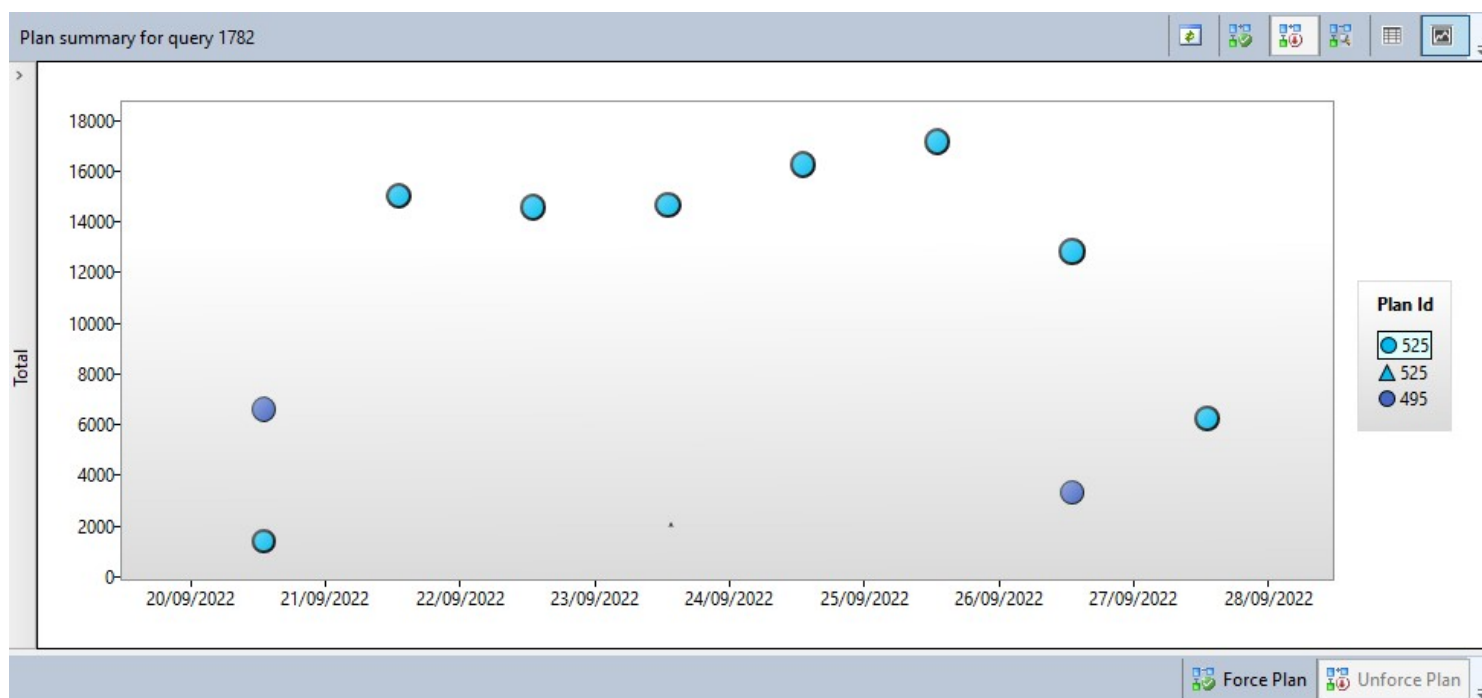
```

MonWiQdsWaitStats
| where TIMESTAMP >= ago (1d)
//| where TIMESTAMP >= datetime(2022-09-20 00:00:00Z)
//| where TIMESTAMP <= datetime(2022-09-21 00:00:00Z)
| where LogicalServerName == "<>"
| where query_hash in ( "0x7AFCBA59FE4C8660" )
| summarize sum(total_query_wait_time_ms) by wait_category, bin(TIMESTAMP, 15m), query_hash
| render timechart

```

Using Query store

Aside from using ASC and Kusto to track Query Execution Plans, Query Store can be used alongside the customer. Below shows a particular Query ID with multiple plans and varying durations/performance. Additionally, the built-in Regressed Queries report in Query Store can be used [Regressed Queries](#).



Sample Troubleshooting

Below is a sample Kusto Query that can be used to demonstrate query plan regression and different behaviours referencing our internal CMS queries/data. Investigation of query plan regressions is explained below over the example output.

```

MonWiQdsExecStats | where AppTypeName == "Control" and AppName contains "cms"
| where originalEventTimestamp > now(-6h)
| extend elapsed_time_millisec = 1.0 * elapsed_time / 1000
| extend interval_start_time_date = interval_start_time / 4294967296
| extend interval_start_time_time = interval_start_time - 4294967296 * interval_start_time_date
| extend interval_start = datetime(1900-1-1) + time(1d) * interval_start_time_date + time(
| extend regressed_plan_id = plan_id
| summarize avg(elapsed_time_millisec), max( interval_start), count() by query_id, regres
| extend regressed_plan_execution_count = count_
| project query_id, regressed_plan_id, avg_elapsed_time_millisec , max_interval_start, regr
| join kind = inner (
    MonWiQdsExecStats
    | where AppTypeName == "Control" and AppName contains "cms"
    | where originalEventTimestamp > now(-6h)
    | extend elapsed_time_millisec1 = 1.0 * elapsed_time / 1000
    | extend interval_start_time_date1 = interval_start_time / 4294967296
    | extend interval_start_time_time1 = interval_start_time - 4294967296 * interval_start_time_date1
    | extend interval_start1 = datetime(1900-1-1) + time(1d) * interval_start_time_date1 +
    | extend new_plan_id = plan_id
    | summarize avg(elapsed_time_millisec1), max( interval_start1), count() by query_id, n
    | extend new_plan_execution_count = count_
    | project query_id, new_plan_id , avg_elapsed_time_millisec1, max_interval_start1, new
) on query_id
| where 2*avg_elapsed_time_millisec < avg_elapsed_time_millisec1 and max_interval_start <

```

query_id	plan_id	avg_elapsed_time	max_interval_start	query_id1	plan_id1	avg_elap
1489444	519798	33397.8181818182	2022-09-25 10:00:00.0000000	1489444	506335	830
1411628	498070	245089	2022-09-25 13:00:00.0000000	1411628	490895	584
1411628	498070	245089	2022-09-25 13:00:00.0000000	1411628	491584	
1411628	490707	85073	2022-09-25 18:00:00.0000000	1411628	490895	584
1411628	490707	85073	2022-09-25 18:00:00.0000000	1411628	490546	104


The example output shows that:

- There are 2 different queries (Query IDs 1489444 and 1411628) which exhibit older, more optimal plans.
- The first row shows that, Query 1489444 has two plans; the old plan was last executed at 10AM and the average elapsed time for this plan (Plan ID 519798) was 33 seconds.
- The new plan (Plan ID 506335) was last executed at 8PM and had an average elapsed time of 83 seconds. The previous execution plan (Plan ID 519798), in this case, is more optimal.

- The 2nd to 5th rows show that Query ID 1411628 has various execution plans. A single query with 4 different plans in the last 6 hours. There are three issues here,
 1. Frequent recompilations: We should investigate what is causing this and fix; follow the section [Frequent Query Recompilation](#).
 2. Plan Regressions: We should find the most effective plan and force it. If you are aware that the query is using various plans and you want to compare them, follow [Capturing Plan Statistics](#).
 3. Non optimal query plan: For query 1411628, the most effective plan is 490707 with 85 sec of execution time; the other 4 plans have execution times of 245 sec, 584 sec, 429 sec and 104 seconds. Even though we have a plan executing in 85 seconds it is still a relatively high cost for one query. In this sample data where we take CMS queries, they should finish within less than 1- second. We need to investigate and look at options to re-write this query.

Mitigation

Depending on the findings and customer scenario, consider the following as potential mitigations:

1. Forcing an execution plan. If you're able to identify that one execution plan is more performant over another, consider [Forcing an Execution Plan](#)  Note that plan forcing is a suitable method to stabilise query performance but the customer should periodically review if the forced execution plan is still the most optimal.
2. Instead of plan forcing, if you are looking to tune a particular query use [Query Tuning](#).
3. SQL Server will cache and reuse execution plans, but in some scenarios where parameter values are passed, the originally cached plan (compiled with certain parameter values) may not fit all parameter variations and can lead to poor query performance. See [Parameter Sniffing](#).

Public Doc Reference

[Plan Forcing](#) 

[High CPU Consuming Troubleshooting reference](#) 

[Regressed Queries - Query Store](#) 

Root Cause Classification

Cases resolved by this TSG should be coded to the following root cause: Root Cause: Azure SQL DB v2\Workload Performance\User Issue/Error\DTU Limit\CPU

How good have you found this content?

