High CPU Utilization

Last updated by | Holger Linke | Jan 19, 2023 at 1:11 AM PST

Contents

Common Solution

This is the common solution article related to high CPU issues that's displayed to customers in the Azure portal when creating a support ticket.

Resolve high CPU utilization in Azure SQL Database

High CPU utilization issues in Azure SQL Database are usually caused by outdated index statistics, missing indexes, query-plan regression, parameter sensitive plan (PSP) issues, poorly designed queries, or increased workloads.

Learn how to resolve high CPU usage by using our diagnostics, reviewing and applying the recommendations, identifying responsible queries, and asking the right questions for narrowing down the cause further. Scan the following headings and see the steps in each section to resolve your issue.

High CPU diagnostics

(runs a check against the resource that was selected on the portal; returns the same/similar insights that are shown to you on the performance troubleshooter in ASC)

Emergency mitigation - scale up for more CPU resources

Consider this step under the following conditions:

- your database is constantly running above 90% of CPU capacity
- your production is severely impacted
- you can't reduce the current workload e.g. by running tasks at an off-peak day or time
- you don't have sufficient time for troubleshooting

If these conditions apply to your situation, then scale the database to a higher service objective or service tier to acquire more allocated resources. Consider to at least double the DTU or CPU capacity (e.g. scale from 200 to 400 or more DTUs, or from 4 to 8 or more vCores). This will reduce the immediate impact and can buy you time for analyzing the cause through the steps from the sections below.

Note: For databases using the DTU purchasing model, a service objective of Standard S3 or higher is strongly recommended for production workloads. Basic, Standard S0, Standard S1 and Standard S2 are best suited for development, testing, and infrequently accessed workloads that aren't sensitive to variable performance. Scale to Standard S3 or higher now if your database is currently running with the Basic, S0, S1, or S2 service objective.

The resource limits for each service tier are documented in the following set of articles:

- DTU model: Single databases and Elastic pools
- vCore model: Single databases and Elastic pools

CPU utilization metrics for the past 24 hours

(displays a chart showing the average percentage of CPU usage for the database for the past 24 hours)
(displays a button to see the top 5 CPU queries on the portal's Query Performance Insights)

Identify currently running queries with high CPU utilization using T-SQL

Run this T-SQL query in the affected database if the issue is occurring at this moment. The query lists the currently running queries sorted by highest CPU usage. It returns the session id, query status, query start time, CPU time in milliseconds, login name, host name, program name, query text, execution plan, and additional details.

The column "query_plan_with_in_flight_statistics" provides you with the actual execution plan including the current, in-query execution statistics. It is a good starting point for investigating non-optimal query execution plans and for further query tuning.

```
-- run in affected user database
SELECT
    req.session_id,
    req.status,
    req.start_time,
    req.cpu_time AS 'cpu_time_ms',
    req.query_hash,
    req.logical_reads,
    req.dop,s.login_name,
    s.host_name,
    s.program_name,
    object_name(st.objectid, st.dbid) as 'object_name',
    REPLACE (REPLACE (
        SUBSTRING (st.text, (req.statement_start_offset/2) + 1,
            ((CASE req.statement_end_offset
             WHEN -1 THEN DATALENGTH(st.text)
            ELSE req.statement_end_offset END - req.statement_start_offset)/2) + 1),
        CHAR(10), ' '), CHAR(13), ' ') AS statement_text,
    qp.query_plan,
    qsx.query_plan as query_plan_with_in_flight_statistics
FROM sys.dm_exec_requests as req
INNER JOIN sys.dm_exec_sessions as s on req.session_id=s.session_id
CROSS APPLY sys.dm_exec_sql_text(req.sql_handle) as st
OUTER APPLY sys.dm_exec_query_plan(req.plan_handle) as qp
OUTER APPLY sys.dm_exec_query_statistics_xml(req.session_id) as qsx
WHERE req.session_id <> @@SPID
ORDER BY req.cpu_time desc;
```

If this doesn't return a clear indication of a cause, then it is also possible that many smaller queries are causing a cumulatively high CPU. In this case, use the query in the next section "Identify queries in the past with high CPU utilization using T-SQL", which can identify this scenario.

Identify queries in the past with high CPU utilization using T-SQL

If the high CPU usage occurred in the past, then Query Store likely has captured a history of queries, execution plans, and runtime statistics for your review. Run the following T-SQL query to list the top 15 queries by CPU usage from Query Store in the previous 24 hours. To adjust the time frame, modify the "DATEADD" parameter in line "rsi.start_time >= DATEADD(HOUR, -24, GETUTCDATE())" to return results from the period when the issue had occurred.

Look for the following two scenarios in the resultset:
(1) Many smaller queries are causing a cumulatively high CPU: the symptoms include a large total CPU, a small average CPU, and a large number of executions.
(2) One or a few queries are consuming most of the CPU: the symptoms include a large total CPU, a large average CPU, and a low number of executions.

```
-- run in affected user database
WITH
AggregatedCPU AS
  (SELECT
    q.query_hash,
    SUM(count_executions) AS total_executions,
    ROUND(SUM(count_executions * avg_cpu_time / 1000.0), 0) AS total_cpu_ms,
    ROUND(SUM(count_executions * avg_cpu_time / 1000.0) / SUM(count_executions), 0) AS avg_cpu_ms,
    ROUND(MAX(rs.max_cpu_time / 1000), 0) AS max_cpu_ms,
    MAX(max_logical_io_reads) AS max_logical_reads,
    COUNT(DISTINCT p.plan_id) AS number_of_distinct_plans,
    COUNT(DISTINCT p.query_id) AS number_of_distinct_query_ids,
    SUM(CASE WHEN rs.execution_type_desc='Aborted' THEN count_executions ELSE 0 END) AS aborted_execution_coun
    SUM(CASE WHEN rs.execution_type_desc='Regular' THEN count_executions ELSE 0 END) AS regular_execution_coun
    SUM(CASE WHEN rs.execution_type_desc='Exception' THEN count_executions ELSE 0 END) AS exception_execution_
    MIN(qt.query_sql_text) AS sampled_query_text
   FROM sys.query_store_query_text AS qt
   INNER JOIN sys.query_store_query AS q ON qt.query_text_id=q.query_text_id
   INNER JOIN sys.query_store_plan AS p ON q.query_id=p.query_id
   INNER JOIN sys.query_store_runtime_stats AS rs ON rs.plan_id=p.plan_id
   INNER JOIN sys.query_store_runtime_stats_interval AS rsi ON rsi.runtime_stats_interval_id=rs.runtime_stats_
   WHERE
     rs.execution_type_desc IN ('Regular', 'Aborted', 'Exception') AND
     rsi.start_time >= DATEADD(HOUR, -24, GETUTCDATE())
   GROUP BY q.query_hash),
OrderedCPU AS
   (SELECT *,
    ROW_NUMBER() OVER (ORDER BY total_cpu_ms DESC, query_hash ASC) AS RN
    FROM AggregatedCPU)
SELECT *
FROM OrderedCPU AS OC
WHERE OC.RN <= 15
ORDER BY total_cpu_ms DESC;
```

If you would like to find more details from the Query Store, also see [Performance tuning sample queries]. This article provides sample queries to get information like [last executed queries], [execution counts], [queries with multiple execution plans], [queries that recently regressed in performance], and others.

Identify top CPU-consuming queries in SQL Server Management Studio (SSMS)

SQL Server Management Studio (SSMS) provides you with a visual interface to several Query Store-based reports. Their predefined views give you the power to discover and tune queries in your workload, and to identify performance bottlenecks in your database, including CPU issues.

The most important reports for resolving CPU issues are:

- **Top Resource-Consuming Queries** which identifies queries with the largest resource consumption (including by CPU time)
- **Regressed Queries** for pinpointing queries that have recently regressed in performance
- **Query Wait Statistics** for analyzing the most active wait categories and contributing queries.

For a practical example, see [Use the Regressed Queries feature] which demonstrates the steps in SSMS and some of the options that are available from the troubleshooting reports.

Identify recent changes in database workload

High CPU utilization doesn't necessarily result from any issues with the query execution, but can be related to an increase in overall database workload. If the queries perform as before but are executed more often, the resource consumption including CPU will be higher than before. In this case, the best mitigation option is to scale the database to a higher service tier or service objective.

The following query shows you a trend over the past 14 days. It relates the number of query executions to the consumed CPU time, the average query duration, and the logical reads. If the execution count increases together with the values for CPU and logical reads, it is an indicator for increasing database workloads.

```
-- run in affected user database
SELECT
    YEAR(start_time) AS 'year', MONTH(start_time) AS 'month', DAY(start_time) AS 'day', DATEPART(hour, start_t
    SUM (rs.count_executions) AS 'execution_count',
    SUM(CASE WHEN rs.execution_type_desc='Regular' THEN count_executions ELSE 0 END) AS 'regular_execution_cou
    SUM(CASE WHEN rs.execution_type_desc='Aborted' THEN count_executions ELSE 0 END) AS 'aborted_execution_cou
    SUM(CASE WHEN rs.execution_type_desc='Exception' THEN count_executions ELSE 0 END) AS 'exception_execution
    ROUND(CONVERT(float, SUM(rs.avg_cpu_time * rs.count_executions)) / NULLIF(SUM(rs.count_executions), 0) * 0
    ROUND(CONVERT(float, SUM(rs.avg_duration * rs.count_executions)) / NULLIF(SUM(rs.count_executions), 0) * 0
    ROUND(CONVERT(float, SUM(rs.avg_logical_io_reads * rs.count_executions)) / NULLIF(SUM(rs.count_executions)
FROM sys.query_store_runtime_stats rs (NOLOCK)
INNER JOIN sys.query_store_runtime_stats_interval i ON rs.runtime_stats_interval_id = i.runtime_stats_interval
-- limit to time window as appropriate:
-- WHERE start_time >= '2023-01-15 09:00:00' AND start_time <= '2023-01-18 18:00:00'
GROUP BY YEAR(start_time), MONTH(start_time), DAY(start_time), DATEPART(hour, start_time)
ORDER BY YEAR(start_time), MONTH(start_time), DAY(start_time), DATEPART(hour, start_time);
```

For an alternate view, you can also compare the percentages for worker threads and sessions (based on the configured service level) in sys.resource_stats (Azure SQL Database). It will show you any scaling operation that might have occurred recently. If the percentages for worker threads and sessions increase together with the CPU percentage, it is an indicator for increasing database workloads.

Run this query in the "master" database and filter on the name of the affected database:

```
-- run in master
-- set the name of the affected database in the Where clause
SELECT
    YEAR(start_time) AS 'year', MONTH(start_time) AS 'month', DAY(start_time) AS 'day', DATEPART(hour, start_t
    MIN(sku) AS 'SKU',
    MIN(dtu_limit) AS 'DTU_limit',
    MIN(cpu_limit) AS 'CPU_limit',
    MAX(max_worker_percent) AS 'worker_percent',
    MAX(max_session_percent) AS 'session_percent',
    MAX(avg_cpu_percent) AS 'CPU_percent'
FROM sys.resource_stats
WHERE database_name = 'yourdatabasename'
-- limit to time window as appropriate:
-- AND start_time >= '2023-01-15 09:00:00' AND start_time <= '2023-01-18 18:00:00'
GROUP BY YEAR(start_time), MONTH(start_time), DAY(start_time), DATEPART(hour, start_time)
ORDER BY YEAR(start_time), MONTH(start_time), DAY(start_time), DATEPART(hour, start_time);
```

Optimize the configured Max Degree of Parallelism (MAXDOP)

The recommended configuration for the maximum degree of parallelism (MAXDOP) in Azure SQL Database is between 1 and 8. Severe performance issues including excessive CPU consumption may occur if MAXDOP is configured outside of the 1...8 range.

Run the following T-SQL command inside your database to check the current MAXDOP configuration and receive further recommendations.

```
-- run in affected user database
declare @maxdop sql_variant;
declare @stmt nvarchar(500), @action nvarchar(500);
SELECT @maxdop = [value] FROM sys.database_scoped_configurations WHERE [name] = 'MAXDOP';
if @maxdop < 1 or @maxdop > 8
    SELECT @stmt = 'MAXDOP is ' + CAST(@maxdop as varchar(3))
    + ' and outside the recommended range of 1...8. ' + CHAR(13) + CHAR(10)
    + 'Change MAXDOP to 8 by running:',
    @action = 'ALTER DATABASE SCOPED CONFIGURATION SET MAXDOP = 8;' ;
else
    SELECT @stmt = 'MAXDOP is ' + cast(@maxdop as varchar(3))
    + ' and within the recommended range of 1...8. ',
    @action = '(no change needed)';
PRINT @stmt;
PRINT @action;
```

Limiting the MAXDOP setting to the 1...8 range reduces the frequency and severity of incidents caused by excessive query parallelism. Parallelism can in general improve workload performance, but can also cause unnecessary resource utilization if it is configured incorrectly. The recommendation is to avoid a MAXDOP setting of "0" even if it doesn't appear to cause problems currently. See Why MAXDOP matters for further background details on this topic, and the discussion on NUMA nodes in SQL Server recommendations.

Update statistics and rebuild fragmented indexes

The query optimizer might produce a non-optimal execution plan because of a missing index, stale statistics, an incorrect estimate of the number of rows to be processed, or an inaccurate estimate of the required memory. If you know the query was executed faster in the past or on another instance, it can help to rebuild indexes and update statistics to reduce CPU utilization and improve performance.

Run the following two queries to see if statistics or indexes have become stale or fragmented. If either or both queries return results, then index and statistics maintenance is recommended. Use the stored procedure from the support blog article How to maintain Azure SQL Indexes and Statistics for this task. The stored procedure has built-in logic to update statistics and to rebuild or reorganize indexes only when needed, thus minimizing the impact on concurrent workloads.

Find stale statistics

This query shows statistics for which the underlying data had been changed since the last "UPDATE STATISTICS" operation ("modification_counter > 0"). It returns the date when those statistics were updated most recently, along with information about the sampling size and the number of data modifications since the last statistics update.

```
-- run in affected user database
SELECT
    ObjectSchema = OBJECT_SCHEMA_NAME(s.object_id),
    ObjectName = object_name(s.object_id),
    s.object_id,
    s.stats_id,
    s.name AS 'stats_name',
    CASE WHEN (s.stats_id > 2 AND s.auto_created = 1) THEN 'AUTOSTATS'
         WHEN (s.stats_id > 2 AND s.auto_created=0) THEN 'STATS' ELSE 'INDEX' END AS type,
    i.type_desc,
    sp.last_updated,
    sp.rows,
    sp.rows_sampled,
    sp.modification_counter
FROM sys.stats s
OUTER APPLY sys.dm_db_stats_properties(s.object_id,s.stats_id) sp
LEFT JOIN sys.indexes i ON sp.object_id = i.object_id AND sp.stats_id = i.index_id
WHERE OBJECT_SCHEMA_NAME(s.object_id) != 'sys'
AND last_updated IS NOT NULL
AND (isnull(sp.modification_counter, 0) > 0 )  -- filter on modifications since last Update Statistics
ORDER BY sp.last_updated ASC;
```

Find Index Fragmentation Level

This query returns all indexes that can benefit from either an index reorganize or index rebuild. It identifies indexes of a certain size that are fragmented more than 5 percent. The maintenance stored procedure from How to maintain Azure SQL Indexes and Statistics can then decide if a reorganize or a rebuild will be the better maintenance operation.

```
-- run in affected user database
SELECT
    idxs.[object_id],
    ObjectSchema = OBJECT_SCHEMA_NAME(idxs.object_id),
    ObjectName = OBJECT_NAME(idxs.object_id),
    IndexName = idxs.name,
    idxs.type,
    idxs.type_desc,
    i.avg_fragmentation_in_percent,
    i.page_count,
    i.index_id,
    i.partition_number,
    i.avg_page_space_used_in_percent,
    i.record_count
FROM sys.indexes idxs
INNER JOIN sys.dm_db_index_physical_stats(DB_ID(), NULL, NULL, NULL, 'SAMPLED') i
    ON i.object_id = idxs.object_id AND i.index_id = idxs.index_id
WHERE idxs.type IN (0 /*HEAP*/, 1/*CLUSTERED*/, 2/*NONCLUSTERED*/, 5/*CLUSTERED COLUMNSTORE*/, 6/*NONCLUSTERED
AND (alloc_unit_type_desc = 'IN_ROW_DATA' /*avoid LOB_DATA or ROW_OVERFLOW_DATA*/
    OR alloc_unit_type_desc IS NULL /*for ColumnStore indexes*/)
AND OBJECT_SCHEMA_NAME(idxs.object_id) != 'sys'
AND page_count > 40
AND avg_fragmentation_in_percent > 5
AND idxs.is_disabled = 0
ORDER BY i.avg_fragmentation_in_percent DESC, i.page_count DESC;
```

Enable Automatic tuning

Non-optimal query performance is often caused by non-optimal query execution plans. With Automatic tuning enabled, Azure SQL Database automatically forces the last known good query plan on your queries. The database engine continuously monitors query performance of the query with the forced plan. If there are performance gains, the database engine will keep using the last known good plan. If performance gains are not detected, the database engine will produce a new plan. In a similar way, it can intelligently manage the indexes in your database if you don't want to tune indexes yourself.

If you haven't enabled automatic tuning yet, use the button below to enable it. Or go to the Azure portal, navigate to your server or your database, and select **Automatic tuning** in the **Intelligent Performance** section of the menu. Enable the "FORCE PLAN" option, and if you haven't tuned your indexes yet, enable the "CREATE INDEX" and "DROP INDEX" options.

(displays button "Enable Automatic Tuning")

If you prefer, you can instead use the ALTER DATABASE T-SQL command to manage automatic tuning:

```
-- configure default tuning options
ALTER DATABASE [database_name] SET AUTOMATIC_TUNING=AUTO

-- configure individual tuning options
ALTER DATABASE [database_name] SET AUTOMATIC_TUNING ( CREATE_INDEX = ON )
ALTER DATABASE [database_name] SET AUTOMATIC_TUNING ( DROP_INDEX = ON )
ALTER DATABASE [database_name] SET AUTOMATIC_TUNING ( FORCE_LAST_GOOD_PLAN = ON )
```

Identifying and adding missing indexes

The database engine in Azure SQL Database is evaluating queries if they could benefit from indexes that don't exist yet. It constantly looks into the execution cost of queries and into potential indexes that could reduce the estimated cost to run a query. The database engine also tracks how often each query plan is executed and what the estimated gap is between the executing query plan and the imagined one where that index existed.

The results are exposed through DMVs that you can query. These allow you to see which changes to your physical database design might improve overall workload cost for a database and its real workload.

Use the following T-SQL query to evaluate potential missing indexes and generate commands to create them:

```
-- run in affected user database
SELECT
    CONVERT (varchar(30), getdate(), 126) AS runtime,
    CONVERT (decimal (28, 1), migs.avg_total_user_cost * migs.avg_user_impact * (migs.user_seeks + migs.user_s
    'CREATE INDEX [missing_index_' + CONVERT (varchar, mig.index_group_handle) + '_' + CONVERT (varchar, mid.i
      + LEFT (PARSENAME(mid.statement, 1), 32) + ']'
      + ' ON ' + mid.statement + ' (' + ISNULL (mid.equality_columns,'')
      + CASE WHEN mid.equality_columns IS NOT NULL AND mid.inequality_columns IS NOT NULL THEN ',' ELSE '' END
      + ISNULL (mid.inequality_columns, '') + ')'
      + ISNULL (' INCLUDE (' + mid.included_columns + ')', '') AS create_index_statement,
    migs.*,
    mid.database_id,
    mid.[object_id]
FROM sys.dm_db_missing_index_groups mig
INNER JOIN sys.dm_db_missing_index_group_stats migs ON migs.group_handle = mig.index_group_handle
INNER JOIN sys.dm_db_missing_index_details mid ON mig.index_handle = mid.index_handle
WHERE CONVERT (decimal (28, 1), migs.avg_total_user_cost * migs.avg_user_impact * (migs.user_seeks + migs.user
ORDER BY estimated_improvement DESC;
```

See Identifying and adding missing indexes for a practical, detailed example. Refer to Tune nonclustered indexes with missing index suggestions for detailed best-practices guidance about combining missing index recommendations with your existing index design.

Resolve queries with non-optimal query execution plans

If you have identified one or more queries that are responsible for your CPU issue, then the next step is to investigate the query's execution plan and look for tuning and improvement options. The article Resolving queries with suboptimal query execution plans gives you some starting points for this task, specifically through these three topics:

- Queries that have parameter sensitive plan (PSP) problems
- Compile activity caused by improper parameterization
- Factors that affect query plan changes

Questions for narrowing down the cause further

When encountering a high-CPU problem, finding answers to the following questions may help you with narrowing down the scope of the issue further:

- Has the issue started suddenly, or was there a steady increase in CPU usage over the past days or weeks?
- If it started suddenly: are you aware of any change or incident that occurred shortly before? Was there an application code update? Was the database scaled around the time, or was there any planned or unplanned failover of the database?
- Have additional users been added to the system, either to an existing application or by hosting a new application? Could this explain additional workloads that haven't been seen previously?
- Do the queries that you identified by the steps in this article relate to a specific application? Are you aware of any recent change to the application?
- Is it possible to move the execution of these queries to off-peak days or times?

Resources

- Resolving queries with suboptimal query execution plans
- Diagnose and troubleshoot high CPU on Azure SQL Database
- Tune applications and databases for performance in Azure SQL Database
- Identify current and previous CPU performance issues using DMVs
- Monitoring and performance tuning in Azure SQL Database and Azure SQL Managed Instance
- Optimize index maintenance to improve query performance and reduce resource consumption
- Monitor performance using Query Store

How good have you found this content?