

Evaluate performance improvements

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/evaluate-performance-improvements/1-introduction>

Introduction

Completed

One of the challenges faced by database administrators is evaluating the changes made to code or data structures on a busy production system. While tuning a single query in isolation provides straightforward metrics such as elapsed time or logical reads, making minor adjustments on a busy system requires a more comprehensive evaluation.

In a production environment, numerous factors can influence the performance of a query or data structure. These include concurrent user activity, system resource utilization, and the overall workload. As a result, you must consider the broader context when assessing the effectiveness of their changes. This involves monitoring system performance over time, analyzing query execution plans, and understanding the interactions between different components of the database.

Additionally, you need to be aware of potential side effects that may arise from their modifications. For instance, a change that improves the performance of one query might inadvertently degrade the performance of another. Therefore, thorough testing and validation are essential to ensure that any adjustments made don't negatively impact the overall system.

Ultimately, the goal is to achieve a balance between optimizing individual queries and maintaining the stability and efficiency of the entire production system. By adopting a comprehensive approach to evaluation, you can ensure that their changes lead to meaningful improvements without compromising the overall performance of the database.

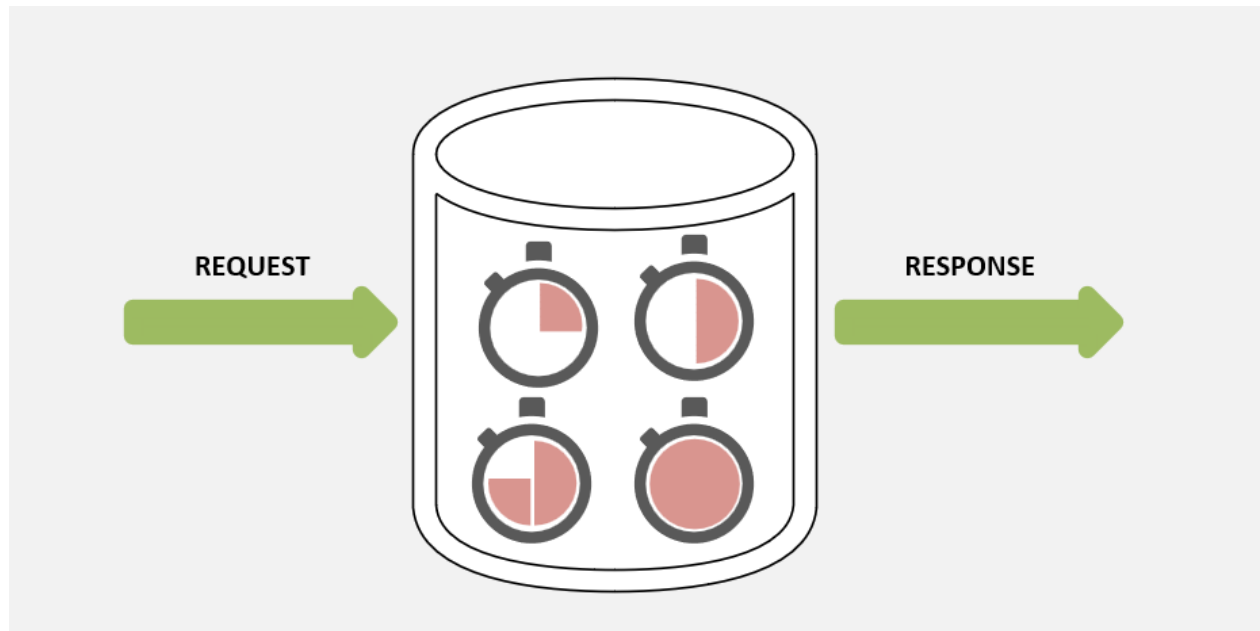
2. Describe wait statistics

Describe wait statistics

Completed

A comprehensive approach to monitoring server performance involves evaluating what the server is waiting on. Wait statistics are intricate, and SQL Server is equipped with hundreds of wait types that monitor each running thread and log what the thread is waiting for.

To effectively detect and troubleshoot SQL Server performance issues, it's essential to understand how wait statistics work and how the database engine utilizes them while processing requests. This knowledge allows you to pinpoint bottlenecks and optimize performance more accurately.



Wait statistics are broken down into three types of waits: **resource waits**, **queue waits**, and **external waits**.

- **Resource waits** occur when a worker thread in SQL Server requests access to a resource that is currently being used by a thread. Examples of resource waits are locks, latches, and disk I/O waits.
- **Queue waits** occur when a worker thread is idle and waiting for work to be assigned. Examples of queue waits are deadlock monitoring and deleted record cleanup.
- **External waits** occur when SQL Server is waiting on an external process like a linked server query to complete. An example of an external wait is a network wait related to returning a large result set to a client application.

You can check `sys.dm_os_wait_stats` system view to explore all the waits encountered by threads that executed, and `sys.dm_db_wait_stats` for Azure SQL Database. The `sys.dm_exec_session_wait_stats` system view lists active waiting sessions.

These system views allow you to get an overview of the performance of the server, and to readily identify configuration or hardware issues. This data is persisted from the time of instance startup, but the data can be cleared as needed to identify changes.

Wait statistics are evaluated as a percentage of the total waits on the server.

	Wait Type	Wait Percentage	AvgWait_Sec
1	REDO_THREAD_PENDING_WORK	24.75	0.1016
2	CXPACKET	17.19	0.0021
3	CXCONSUMER	12.24	0.0090
4	PARALLEL_REDO_TRAN_TURN	10.60	0.0029
5	SOS_SCHEDULER_YIELD	10.06	0.0022
6	BPSORT	3.80	0.0126
7	PAGEIOLATCH_SH	3.60	0.0029
8	BACKUPIO	2.01	0.0110
9	HTDELETE	1.83	0.1573
10	LATCH_EX	1.81	0.0047

The result of this query from `sys.dm_os_wait_stats` shows the wait type, and the aggregation of percent of time waiting (*Wait Percentage* column) and the average wait time in seconds for each wait type.

In this case, the server has Always On Availability Groups in place, as indicated by the **REDO_THREAD_PENDING_WORK** and **PARALLEL_REDO_TRAN_TURN** wait types. The relatively high percentage of **CXPACKET** and **SOS_SCHEDULER_YIELD** waits indicates that this server is under some CPU pressure.

As DMVs provide a list of wait types with the highest time accumulated since the last SQL Server startup, collecting and storing wait statistic data periodically could help you understand and correlate performance problems with other database events.

Considering that DMVs provide you with a list of wait types with the highest time accumulated since the last SQL Server startup, collecting and storing wait statistics periodically might help you understand and correlate performance problems with other database events.

There are several types of waits available in SQL Server, but some of them are common.

- **RESOURCE_SEMAPHORE**—indicates that queries are waiting for memory to become available, often due to excessive memory grants to certain queries. This issue typically

manifests as long query runtimes or even time-outs. Causes of these wait types can include out-of-date statistics, missing indexes, and high query concurrency.

- **LCK_M_X**—frequently indicates a blocking problem. This issue can be resolved by changing to the `READ COMMITTED SNAPSHOT` isolation level, optimizing indexing to reduce transaction times, or improving transaction management within T-SQL code.
- **PAGEIOLATCH_SH**—this wait type can indicate issues with indexes or the absence of useful indexes, causing SQL Server to scan excessive amounts of data. Alternatively, if the wait count is low but the wait time is high, it may suggest storage performance problems. You can observe this behavior by analyzing the data in the `waiting_tasks_count` and `wait_time_ms` columns in the `sys.dm_os_wait_stats` system view to calculate the average wait time for a given wait type.
- **SOS_SCHEDULER_YIELD**—this wait type can indicate high CPU utilization, which is correlated with either high number of large scans, or missing indexes, and often with high numbers of **CXPACKET** waits.
- **CXPACKET**—A high occurrence of this wait type can indicate improper configuration. Before SQL Server 2019, the default setting for the `max degree of parallelism (MAXDOP)` was to use all available CPUs for queries. Additionally, the cost threshold for parallelism was set to 5, which could cause small queries to be executed in parallel, limiting throughput. To reduce this wait type, you can lower the MAXDOP setting and increase the cost threshold for parallelism. However, the **CXPACKET** wait type can also indicate high CPU utilization, which is typically resolved through index tuning.
- **PAGEIOLATCH_UP**—This wait type on data pages 2:1:1 can indicate TempDB contention on Page Free Space (PFS) data pages. Each data file has one PFS page per 64 MB of data. This wait is typically caused by only having one TempDB file, as prior to SQL Server 2016, the default behavior was to use one data file for TempDB. The [best practice for TempDB](#) is to use one file per CPU core, up to eight files. It's also important to ensure your TempDB data files are the same size and have the same autogrowth settings to ensure they're used evenly. SQL Server 2016 and higher control the growth of TempDB data files to ensure they grow in a consistent, simultaneous fashion.

In addition to the DMVs mentioned earlier, the [Query Store](#) also tracks waits associated with specific queries. Although the waits data tracked by the Query Store isn't as granular as the data in the DMVs, it still provides a useful overview of what a query is waiting on.

3. Tune and maintain indexes

Tune and maintain indexes

Completed

The most common (and most effective) method for tuning T-SQL queries is to evaluate and adjust your indexing strategy. Properly indexed databases perform fewer IOs to return query results, and with fewer IOs there's reduced pressure on both the IO and storage systems. Reducing IO even allows for better memory utilization. Keep in mind the read/write ratio of your queries.

A heavy write workload may indicate that the cost of writing rows to extra indexes isn't of much benefit. An exception would be if the workload performs mainly updates that also need to do *lookup* operations. Update operations that do lookups can benefit from extra indexes or columns added to an existing index. Your goal should always be to get the most benefit out of the smallest number of indexes on your tables.

A common performance tuning approach is as follows:

- Evaluate existing index usage using `sys.dm_db_index_operational_stats` and `sys.dm_db_index_usage_stats`.
- Consider eliminating unused and duplicate indexes, but this should be done carefully. Some indexes may only be used during monthly/quarterly/annual operations, and may be important for those processes. You may also consider creating indexes to support those operations just before the operations are scheduled, to reduce the overhead of having otherwise unused indexes on a table.
- Review and evaluate expensive queries from the Query Store, or Extended Events capture, and work to manually craft indexes to better serve those queries.
- Create the indexes in a nonproduction environment, and test query execution and performance and observe performance changes. It's important to note any hardware differences between your production and nonproduction environments, as the amount of memory and the number of CPUs could affect your execution plan.
- After testing carefully, implement the changes to your production system.

Verify the column order of your indexes—the leading column drives column statistics and usually determines whether the optimizer chooses the index. Ideally, the leading column is selective and used in the `WHERE` clause of many of your queries. Consider using a change control process for tracking changes that could affect application performance. Before dropping an index, save the code in your source control, so the index can be quickly recreated if an infrequently run query requires the index to perform well.

Finally, columns used for **equality comparisons** should precede columns used for **inequality comparisons** and that columns with greater selectivity should precede columns with fewer distinct values.

Resumable index

Resumable index allows index maintenance operations to be paused, or take place in a time window, and be resumed later. A good example of where to use resumable index operations is to reduce the impact of index maintenance in a busy production environment. You can then perform rebuild operations during a specific maintenance window giving you more control over the process.

Furthermore, creating an index for a large table can negatively affect the performance of the entire database system. The only way to fix this issue in versions prior to SQL Server 2019 is to kill the index creation process. Then you have to start the process over from the beginning if the system rolls back the session.

With resumable index, you can pause the build and then restart it later at the point it was paused.

The following example shows how to create a resumable index:

```
-- Creates a nonclustered index for the Customer table

CREATE INDEX IX_Customer_PersonID_ModifiedDate
    ON Sales.Customer (PersonID, StoreID, TerritoryID, AccountNumber, ModifiedDate)
WITH (RESUMABLE=ON, ONLINE=ON)
GO
```

In a query window, pause the index operation:

```
ALTER INDEX IX_Customer_PersonID_ModifiedDate ON Sales.Customer PAUSE
GO
```

This statement uses the `PAUSE` clause to temporarily stop the creation of the resumable online index.

You can check the current execution status for a resumable online index by querying the `sys.index_resumable_operations` system view.

Note

Resumable index is only supported with online operations.

4. Understand query hints

Understand query hints

Completed

Query hints are options or strategies that can be applied to enforce the query processor to use a particular operator in the execution plan for `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements. Query hints override any execution plan the query processor might select for a given query with the `OPTION` clause.

In most cases, the query optimizer selects an efficient execution plan based on the indexes, statistics, and data distribution. Database administrators rarely need to intervene manually.

You can change the execution plan of the query by adding query hints to the end of the query. For example, if you add `OPTION (MAXDOP <integer_value>)` to the end of a query that uses a single CPU, the query may use multiple CPUs (parallelism) depending on the value you choose. Or, you can use `OPTION (RECOMPILE)` to ensure that the query generates a new, temporary plan each time it's executed.

```
--With maxdop hint
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE UnitPrice < $5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (MAXDOP 2)
GO

--With recompile hint
SELECT City
FROM Person.Address
WHERE StateProvinceID=15 OPTION (RECOMPILE)
GO
```

Although query hints may provide a localized solution to various performance-related issues, you should avoid using them in production environment for the following reasons.

- Having a permanent query hint on your query can result in structural database changes that would be beneficial to that query not being applicable.
- You can't benefit from new and improved features in subsequent versions of SQL Server if you bind a query to a specific execution plan.

However, there are several query hints available on SQL Server, which are used for different purposes. Let's discuss a few of them below:

- **FAST <integer_value>** —retrieves the first <integer_value> number of rows while continuing query execution. It works better with small data sets and low value for fast query hint. As row count is increased, query cost becomes higher.
- **OPTIMIZE FOR** —provides instructions to the query optimizer that a particular value for a local variable should be used when a query is compiled and optimized.
- **USE PLAN** —the query optimizer uses a query plan specified by the *xml_plan* attribute.
- **RECOMPILE** —creates a new, temporary plan for the query and discards it immediately after the query is executed.
- **{ LOOP | MERGE | HASH } JOIN** —specifies all join operations are performed by **LOOP JOIN**, **MERGE JOIN**, or **HASH JOIN** in the whole query. The optimizer chooses the least expensive join strategy from among the options if you specify more than one join hint.
- **MAXDOP <integer_value>** —overrides the max degree of parallelism value of *sp_configure*. The query specifying this option also overrides the Resource Governor.

You can also apply multiple query hints in the same query. The following example uses the **HASH GROUP** and **FAST <integer_value>** query hints in the same query.

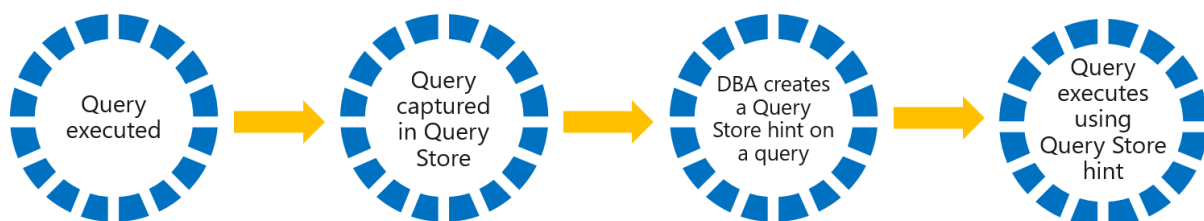
```
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE UnitPrice < $5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (HASH GROUP, FAST 10);
GO
```

To learn more about query hints, see [Hints \(Transact-SQL\)](#).

Query Store hints

[Query Store hints](#) provides a simple method for shaping query plans without modifying application code.

Query Store hints are useful when the query optimizer doesn't generate an efficient execution plan, and when the developer or DBA can't modify the original query text. In some applications, the query text may be hardcoded or automatically generated.



To use Query Store hints, you need to identify the Query Store *query_id* of the query statement you wish to modify through Query Store catalog views, built-in Query Store reports, or Query Performance Insight for Azure SQL Database. Then, execute `sp_query_store_set_hints` with the *query_id* and query hint string you wish to apply to the query.

The following example shows how to obtain the `query_id` for a specific query and then use it to apply the `RECOMPILE` and `MAXDOP` hints to the query.

```
SELECT q.query_id, qt.query_sql_text
FROM sys.query_store_query_text qt
     INNER JOIN sys.query_store_query q
           ON qt.query_text_id = q.query_text_id
WHERE query_sql_text like N'%ORDER BY CustomerName DESC%'
     AND query_sql_text not like N'%query_store%'
GO

--Assuming the query_id returned by the previous query is 42
EXEC sys.sp_query_store_set_hints @query_id= 42, @query_hints = N'OPTION(RECOMPILE
GO
```

There are a few scenarios where Query Store hints can help with query-level performance issues.

- Recompile a query on each execution.
- Limit the maximum degree of parallelism for a statistic update operation.
- Use a Hash join instead of a Nested Loops join.
- Use compatibility level 110 for a specific query while keeping the database at the current compatibility.

For more information about Query Store hints, see [Query Store hints](#).

5. Explore performance scenarios

<https://learn.microsoft.com/en-us/training/modules/evaluate-performance-improvements/4b-explore-performance>

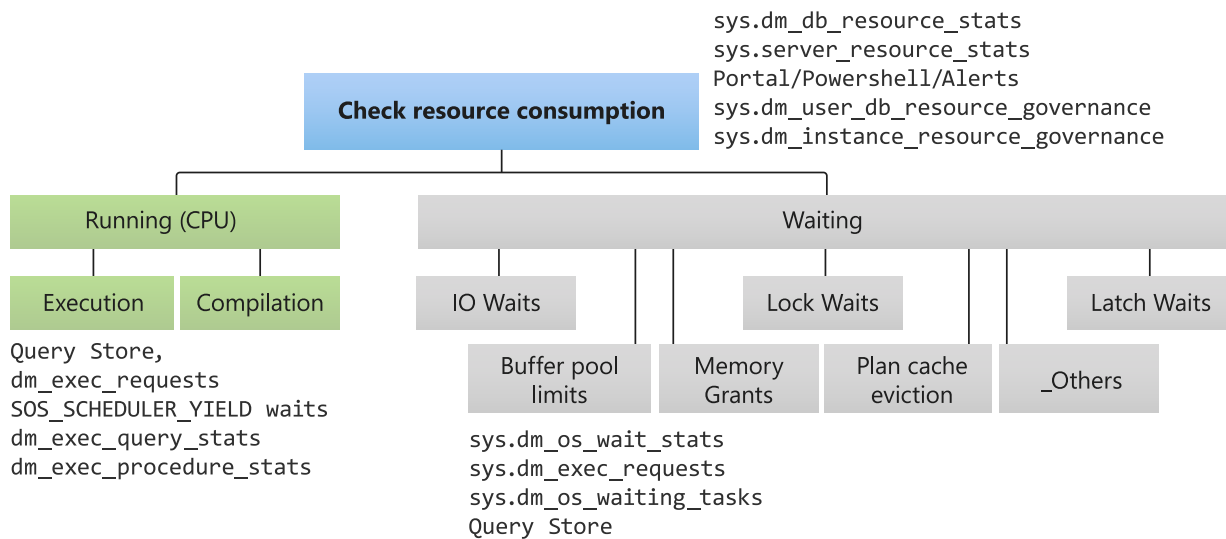
Explore performance scenarios

Completed

To decide how to use performance tools and capabilities, it's important to look at performance for Azure SQL through scenarios.

Understand common performance scenarios

A common technique for SQL Server performance troubleshooting is to examine whether a performance problem is **Running** (high CPU) or **Waiting** (waiting on a resource). The following diagram shows a decision tree to determine if a SQL Server performance issue is running or waiting, and how to use performance tools to determine the cause and solution.



First, look at overall resource usage. For a standard SQL Server deployment, you might use tools such as Performance Monitor in Windows or top in Linux. For Azure SQL, you can use the following methods:

- Azure portal/PowerShell/alerts

Azure Monitor has integrated metrics to view resource usage for Azure SQL. You can also set up alerts to look for resource usage conditions.

- `sys.dm_db_resource_stats`

For Azure SQL Database, you can look at this DMV to see CPU, memory, and I/O resource usage for the database deployment. This DMV takes a snapshot of this data every 15 seconds.

- `sys.server_resource_stats`

This DMV behaves just like `sys.dm_db_resource_stats`, but it's used to see resource usage for the SQL Managed Instance for CPU, memory, and I/O. This DMV also takes a snapshot every 15 seconds.

- `sys.dm_user_db_resource_governance`

For Azure SQL Database, this DMV returns the actual configuration and capacity settings used by resource governance mechanisms in the current database or elastic pool.

- `sys.dm_instance_resource_governance`

For Azure SQL Managed Instance, this DMV returns similar information as `sys.dm_user_db_resource_governance`, but for the current SQL Managed Instance.

Running

If you have determined the problem is high CPU utilization, this is called a running scenario. A running scenario can involve queries that consume resources through compilation or execution. Use the following tools for further analysis:

- Query Store

Use the Top Consuming Resource reports in SSMS, Query Store catalog views, or Query Performance Insight in the Azure portal (Azure SQL Database only) to find which queries are consuming the most CPU resources.

- `sys.dm_exec_requests`

Use this DMV in Azure SQL to get a snapshot of the state of active queries. Look for queries with a state of `RUNNABLE` and a wait type of `SOS_SCHEDULER_YIELD` to see if you have enough CPU capacity.

- `sys.dm_exec_query_stats`

This DMV can be used much like Query Store to find top resource consuming queries. It's only available for query plans that are cached, whereas Query Store provides a persistent historical record of performance. This DMV also allows you to find the query plan for a cached query.

- `sys.dm_exec_procedure_stats`

This DMV provides information much like `sys.dm_exec_query_stats`, except the performance information can be viewed at the stored procedure level.

After you determine what query or queries are consuming the most resources, you might have to examine whether you have enough CPU resources for your workload. You might debug query plans with tools like lightweight query profiling, SET statements, Query Store, or extended events tracing.

Waiting

If your problem doesn't appear to be a high CPU resource usage, it might be that the performance problem involves waiting on a resource. Scenarios involving waiting on resources include:

- I/O waits
- Lock waits
- Latch waits
- Buffer pool limits
- Memory grants
- Plan cache eviction

To perform analysis on waiting scenarios, you'd typically look at the following tools:

- `sys.dm_os_wait_stats`

Use this DMV to see the top wait types for the database or instance. This can guide you on what action to take next, depending on the top wait types.

- `sys.dm_exec_requests`

Use this DMV to find specific wait types for active queries to see what resource they're waiting on. This can be a standard blocking scenario, waiting on locks from other users.

- `sys.dm_os_waiting_tasks`

You can use this DMV to find wait types for a particular task for a specific query that is currently executing, perhaps to see why it's taking longer than normal.

`sys.dm_os_waiting_tasks` contains the live wait stats that `sys.dm_os_wait_stats` aggregates over time.

- Query Store

Query Store provides reports and catalog views that show an aggregation of the top waits for query plan execution. It's important to know that a wait of **CPU** is equivalent to a *running* problem.

Scenarios specific to Azure SQL

There are some performance scenarios, both running and waiting, that are specific to Azure SQL. These include log governance, worker limits, waits encountered for Business Critical service tiers, and waits specific to a Hyperscale deployment.

Log governance

Azure SQL can use log rate governance to enforce resource limits on transaction log usage. You might need this enforcement to ensure resource limits and to meet promised SLA. Log governance might be seen from the following wait types:

- `LOG_RATE_GOVERNOR` : waits for Azure SQL Database
- `POOL_LOG_RATE_GOVERNOR` : waits for Elastic Pools
- `INSTANCE_LOG_GOVERNOR` : waits for Azure SQL Managed Instance
- `HADR_THROTTLE_LOG_RATE*` : waits for Business Critical and geo-replication latency

Worker limits

SQL Server uses a worker pool of threads but has limits on the maximum number of workers. Applications with a large number of concurrent users might approach the worker limits enforced for Azure SQL Database and SQL Managed Instance:

- Azure SQL Database has limits based on service tier and size. If you exceed this limit, a new query receives an error.

- At the current time, SQL Managed Instance uses `max worker threads`, so workers past this limit might see `THREADPOOL` waits.

Business Critical HADR waits

If you use a Business Critical service tier, you might unexpectedly see the following wait types:

- `HADR_SYNC_COMMIT`
- `HADR_DATABASE_FLOW_CONTROL`
- `HADR_THROTTLE_LOG_RATE_SEND_RECV`

Even though these waits might not slow down your application, you might not be expecting to see these. They're normally specific to using an Always On availability group. Business Critical tiers use availability group technology to implement SLA and availability features of a Business Critical service tier, so these wait types are expected. Long wait times might indicate a bottleneck such as I/O latency or replica behind.

Hyperscale

The Hyperscale architecture can result in some unique wait types that are prefixed with **RBIO** (a possible indication of log governance). In addition, DMVs, catalog views, and extended events are enhanced to show metrics for page server reads.

In the next exercise, you'll learn how to monitor and solve a performance problem for Azure SQL by using the tools and knowledge you've gained in this unit.

6. Exercise: Isolate problem areas in poorly performing queries

<https://learn.microsoft.com/en-us/training/modules/evaluate-performance-improvements/5-exercise-isolate-problem-areas-poor-performing-queries>

Exercise: Isolate problem areas in poorly performing queries

Completed

In this exercise, you'll learn how to isolate problem areas in poorly performing queries in a SQL Database using SSMS.

Note

To complete this exercise, you'll need an [Azure subscription](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

7. Module assessment

<https://learn.microsoft.com/en-us/training/modules/evaluate-performance-improvements/6-knowledge-check>

Module assessment

Completed

8. Summary

<https://learn.microsoft.com/en-us/training/modules/evaluate-performance-improvements/7-summary>

Summary

Completed

In this module, you learned about monitoring server performance using SQL Server's wait statistics, which include resource waits, queue waits, and external waits. You have also learned how to use system views like `sys.dm_os_wait_stats` and `sys.dm_db_wait_stats` to get an overview of server performance and identify potential issues. The module also covered the use of Dynamic Management Views (DMVs) to understand and correlate performance problems with other database events. Additionally, you learned about common wait types and how the Query Store tracks waits associated with specific queries.

The main takeaways from this module include understanding how to tune T-SQL queries by evaluating and adjusting your indexing strategy. You learned that proper indexing can reduce IOs, improve memory utilization, and ease pressure on IO and storage systems. The module also discussed the importance of column order in indexes and the use of resumable index for large tables. Furthermore, you learned about query hints and their potential impact on database structure and performance. Lastly, the module covered how to optimize Azure SQL performance by identifying

whether a performance issue is due to high CPU usage or waiting on a resource, and using appropriate tools and methods to diagnose and resolve these issues.

Additional Reading

- [Optimizing Azure SQL Performance](#)
- [Best practices for managing the Query Store](#)
- [Clustered and nonclustered indexes](#)