**Describe wait statistics**

One holistic way of monitoring server performance is to evaluate what the server is waiting on. Wait statistics are complex, and SQL Server is instrumented with hundreds of waiting types, which monitors each running thread and logs what the thread is waiting on.

Detecting and troubleshooting SQL Server performance issues require an understanding of how wait statistics work, and how the database engine uses them while processing a request.

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 resources wait are locks, latches, and disk I/O waits.
* **Queue waits** occur when a worker thread is idle and waiting for work to be assigned. Example 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 the DBA 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.

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**—this wait type is indicative of queries waiting on memory to become available, and may indicate excessive memory grants to some queries. This problem is typically observed by long query runtimes or even time outs. These wait types can be caused by out-of-date statistics, missing indexes, and excessive query concurrency.
* **LCK\_M\_X**—frequent occurrences of this wait type can indicate a blocking problem, that can be solved by either changing to the READ COMMITTED SNAPSHOT isolation level, or making changes in indexing to reduce transaction times, or possibly better transaction management within T-SQL code.
* **PAGEIOLATCH\_SH**—this wait type can indicate a problem with indexes (or a lack of useful indexes), where SQL Server is scanning too much data. Alternatively, if the wait count is low, but the wait time is high, it can indicate storage performance problems. You can observe this behavior by analyzing the data in the *waiting\_tasks\_count* and the *wait\_time\_ms* columns in the sys.dm\_os\_wait\_stats system view, to calculate an 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**—if this wait type is high it can indicate improper configuration. Prior to SQL Server 2019, the max degree of parallelism default setting is to use all available CPUs for queries. Additionally, the cost threshold for parallelism setting defaults to 5, which can lead to small queries being executed in parallel, which can limit throughput. Lowering MAXDOP and increasing the cost threshold for parallelism can reduce this wait type, but 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 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 aforementioned DMVs, the Query Store also tracks waits associated with a given query. However, waits data tracked by Query Store isn't tracked at the same granularity as the data in the DMVs, but it can provide a nice overview of what a query is waiting on.

# Tune and maintain indexes

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 index(s) in a non-production environment, and test query execution and performance and observe performance changes. It's important to note any hardware differences between your production and non-production 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 will choose the index. Ideally, the leading column will be 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:

SQLCopy

-- 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:

SQLCopy

ALTER INDEX IX\_Customer\_PersonID\_ModifiedDate ON Sales.Customer PAUSE

GO

The statement above 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.

# Understand query hints

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.

SQLCopy

--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 will use 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.

SQLCopy

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)](https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query).

## Query Store hints (in preview)

The Query Store hints feature in Azure SQL Database 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 example below 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.

SQLCopy

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, MAXDOP 1)'

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.

**Note**

Query Store hints are also supported by SQL Managed Instance.

Additional references

Query store Hints: [Query Store hints - SQL Server | Microsoft Learn](https://learn.microsoft.com/en-us/sql/relational-databases/performance/query-store-hints?view=sql-server-ver16)